Improving Short DNA Sequence Alignment with Parallel Computing

Darren Peters

Computer Science

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University

St. Catharines, Ontario

Abstract

Variations in different types of genomes have been found to be responsible for a large degree of physical diversity such as appearance and susceptibility to disease. Identification of genomic variations is difficult and can be facilitated through computational analysis of DNA sequences. Newly available technologies are able to sequence billions of DNA base pairs relatively quickly. These sequences can be used to identify variations within their specific genome but must be mapped to a reference sequence first. In order to align these sequences to a reference sequence, we require mapping algorithms that make use of approximate string matching and string indexing methods. To date, few mapping algorithms have been tailored to handle the massive amounts of output generated by newly available sequencing technologies. In order to handle this large amount of data, we modified the popular mapping software BWA to run in parallel using OpenMPI. Parallel BWA matches the efficiency of multithreaded BWA functions while providing efficient parallelism for BWA functions that do not currently support multithreading. Parallel BWA shows significant wall time speedup in comparison to multithreaded BWA on high-performance computing clusters, and will thus facilitate the analysis of genome sequencing data.

Contents

List of Tables

List of Figures

1 Introduction

The recent evolvement of DNA sequencing technologies into what are known as next-generation sequencing (NGS) technologies has resulted in the need for improved computer algorithms to accommodate the massive amounts of data created by these high-throughput technologies. NGS platforms such as 454 (Roche) [47], Illumina (Solexa) [3] and SOLiD (ABI) [31] can sequence billions of base pairs per run of the instrument [31]. These sequenced base pairs are of little value unless they can be arranged or aligned in some meaningful way, usually by being aligned or mapped to a reference genome pertaining to the species from which the sequences originated. Once aligned to a reference genome, the results can be analyzed to determine sequence variations such as single nucleotide polymorphisms and structural variations such as insertions, deletions, and translocations. These variations can cause physical diversity such as appearance and susceptibility to diseases. These variations are essentially what make each human (in the case of the human genome) physically and physiologically unique. It is clear that being able to efficiently determine genetic variations from the large amount of sequences generated by high-throughput sequencing technologies will go a long way towards advancing our knowledge regarding human health.

Efficient determination of genetic variants from billions of DNA sequences can be facilitated through computational analysis. This is a difficult task, due to the short length of the generated sequences and the lack of order in the production of the sequences. Given one short sequence, one can not easily determine from which part of its respective genome the sequence came. The next sequence analysed does not necessarily follow the previous sequence within the genome, which results in having to search the entire genome for the locations of each sequence. Computer algorithms known as sequence alignment software

aim to determine the locations within the reference genome of the sequences generated in a relatively short amount of time. These algorithms must balance speed and accuracy, as improvements to one almost always come at the cost of the other. These algorithms make use of approximate string matching and string indexing in order to efficiently align sequences. Approximate string matching and string indexing are the two main computational techniques used in DNA sequence alignment. The study of approximate string matching details the attempts of matching one sequence of characters to another, longer sequence of characters with some errors allowed. If we think about DNA sequences as a sequence of characters comprised of the letters A, C, G, and T, it is obvious that approximate string matching has an application in DNA sequence alignment. The field of string indexing attempts to provide the ability to efficiently store and search across extremely large texts. When the size of the human genome is taken into account at approximately 3.2 billion base pairs (bp), the use of string indexing in DNA sequence alignment is obvious.

Until recently, most sequence alignment software has been inadequate to address the massive amount of data generated by new high-throughput technologies and there has been great need of better software that can effectively handle this increase in production. While new software such as BWA [26], SOAP2 [30], and Bowtie [17] are able to efficiently align short DNA sequences, NGS platforms are evolving very rapidly, pushing the sequencing capacity at a dramatic speed. For example, platforms based on single molecular sequencing, such as Helicos [15] and PacBio [10], are now able to offer a throughput of billions of sequence reads daily. Such new level of sequencing capacity calls for further speedup in the sequence alignment step. To address this challenge, we have to inevitably go with high performance computing using either input file splitting or parallel computing, since speed improvement via algorithm improvement may be limited. In comparison with the input splitting, in which extremely large data-sets are first split into a large number of smaller

sets, with each aligned individually by distributing them over large clusters followed by concatenating their results via shell scripts, it is proposed that parallel computing should offer better reduction of wall-time and more efficient workflow. Parallel computing is a branch of computer science that involves the execution of an algorithm on multiple processors simultaneously. Traditional computers can only make use of one processor at any single point in time, while parallel computers can make use of many processors at any single point in time. The opportunity for algorithm speedup is obvious if the algorithm can be expertly designed to run on multiple processors simultaneously. It would be of great benefit to the DNA sequencing community if one would be able to take advantage of the opportunities offered by parallel computing.

This thesis covers relevant background information regarding DNA sequence alignment and the work undertaken in using parallel computing to speed up the popular sequence alignment software BWA. Detailed explanations of the work done in approximate string matching and string indexing are given, as well as a detailed look into the main method used by BWA, approximate matching with the Burrows-Wheeler transform. A more in-depth look into the biological side of DNA sequence alignment is also taken. An overview in parallel computing is also provided, followed by the parallel approaches attempted towards improving the speed of BWA. It will be shown that attempts in implementing a parallel version of BWA resulted in, at the cost of processor efficiency, improvements to the sequential version and can speed up the alignment of DNA sequences generated by the new high-throughput technologies.

2 Background

The background section will cover the knowledge required for a basic understanding of how BWA functions and a general overview of parallel computing so that the parallelization process can be followed. It will cover such knowledge in the form of a literature review, starting from the most primitive solutions and working towards the more complex and efficient solutions used in BWA.

2.1 Approximate String Matching

Approximate string matching is the study of a class of computer science problems, which, in their most basic form, attempt to find the location within a text where a given pattern occurs. Since the goal is to find an approximate match, allowance for a maximum specified number of errors in each match is provided [35]. The concept of an error is variable, and is defined depending on the application. A model or definition of what constitutes an error is known as a *distance function.* In the case of DNA re-sequencing, an error can be considered an insertion, a deletion, or a substitution in the pattern. A substitution occurs when one character in the pattern is replaced by another. This distance function is known as *edit distance* [35]. An example of edit distance can be illustrated by attempting to match the word "process" to the word "progress". We can substitute the 'c' in process for a 'g', and insert an 'r' after the 'g' in order to transform "process" into "progress". Since we require two edit operations in order to make the transformation, the edit distance between these two words is two. Other distance functions exist such as Hamming distance and are frequently used in other applications. Hamming distance allows only substitutions in the text, so insertions and deletions are not allowed. However, since DNA re-sequencing uses edit distance, all concepts and examples throughout the remainder of this paper can be assumed

to use edit distance. We will look at a few different approaches to solving the approximate string matching problem, starting with the oldest solution, dynamic programming.

2.1.1 Dynamic Programming

The first solution to the approximate string matching problem was proposed by Sellers in 1980 [40]. Other authors had previously used a similar algorithm to compute edit distance, but Sellers was the first to use it as an approximate matching algorithm. The solution uses dynamic programming and although it is not very efficient, it is very flexible in that it is easy to map the solution to many different distance functions [35]. The solution originally consisted of an algorithm that computes the edit distance between two words, p and t. Throughout the remainder of this paper, we can assume the word p to represent the pattern we are matching to the larger text, known as t. The number of characters in p is known as the *length* of p, and will be denoted as m. The number of characters in (or length of) t is known as n. When allowing errors, we define the maximum edit distance allowed as k. When discussing approximate string matching algorithms, we refer to algorithms that do not pre-process the text or pattern as *online algorithms.*

Computing Edit Distance The algorithm used to calculate edit distance does so by filling an n by m matrix known as C, where $C_{i,j}$ represents the minimum number of edit operations needed to match the first i characters of p to the first j characters of t. The formula to do this is well-known and given below:

$C_{i,0} = i$

$C_{0,j} = j$

$C_{i,j} = \text{if } (x_i = y_j) \text{ then } C_{i-1,j-1}$

$\qquad\qquad\qquad\qquad \text{else } 1 + min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$

When $i = m$ and $j = n$, the formula will produce the edit distance between p and t. The top two lines represent the edit distance between p and an empty string and t and an empty string, respectively. It is clear that i (or j) deletions are required on the first i (or j) characters of the non-empty string to convert it to the empty string. The third line works as follows. We assume all of the edit distances for the shorter substrings have already been calculated. If the i^{th} character of p equals the j^{th} character of t, there are no edit operations required to transform the shorter substring into the current substring, so we assign it the current edit distance. If the two characters are not equal, we require an edit operation, hence the increase in edit distance. At this point, we can either delete a character from the pattern, insert a character into the pattern, or substitute a character within the pattern. These moves represent the $\min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$ part of the formula. The *min* exists as we obviously wish to perform the fewest number of edit operations possible. It is quite a simple formula, however this just computes edit distance between two words. It is not a search algorithm and thus does not recognize patterns within larger texts. For example, if we had a pattern of *hi* and a text of *highlander*, using just the above algorithm would return an edit distance of eight (inserting 'g', 'h', 'l', 'a', etc), even though the pattern exists exactly within the text. An example is given in Figure 1(a), where k is equal to two.

Searching on a Text In order to adapt the above algorithm to search over a text, we must allow any character within the text to be the potential starting point for a match. Thus we set $C_{0,j} = 0$ for all $j \in \{0, 1, 2, ..., n\}$. This allows the pattern to start with an edit distance of zero at any point in the text. We then process each character in t sequentially, updating an entire column to $C'_{0..m}$ for each text character encountered by the following formula:

		p	r	o	g	r	e	s	s
	0	1	2	3	4	5	6	7	8
p	1	0	1	2	3	4	5	6	7
r	2	1	0	1	2	3	4	5	6
o	3	2	1	0	1	2	3	4	5
c	4	3	2	1	1	2	3	4	5
e	5	4	3	2	2	2	2	3	4
s	6	5	4	3	3	3	3	2	3
s	7	6	5	4	4	4	4	3	2

(a) Non-Search Version

		p	r	o	g	r	e	s	s
	0	0	0	0	0	0	0	0	0
p	1	0	1	1	1	1	1	1	1
r	2	1	0	1	2	1	2	2	2
o	3	2	1	0	1	2	2	3	3
c	4	3	2	1	1	2	3	3	4
e	5	4	3	2	2	2	2	3	4
s	6	5	4	3	3	3	3	2	3
s	7	6	5	4	4	4	4	3	2

(b) Search Version

Figure 1: Comparing the two algorithms for pattern 'process' over the text 'progress'

For all $i \in \{0, 1, 2, ..., m\}$:

$$C'_i = \text{if } (P_i = T_j) \quad \text{then} \quad C_{i-1}$$
$$\text{else} \quad 1 + \min(C'_{i-1}, C_i, C_{i-1})$$

The pattern approximately occurs at position j in the text where $C_{m,j}$ is less than k. An example is given in Figure 1(b), where k is equal to two. It is seen that the pattern occurs in the text ending at the last text position.

Complexity of the Dynamic Programming Approach The standard dynamic programming approach is not a very efficient solution. Its worst case time complexity is $O(mn)$, as we need to compute a cell value for each text and pattern character combination. This is the worst complexity of all the approximate string matching solutions we will see. Its space complexity is $O(m)$, as we only need to store the current and previous columns in order to execute the algorithm. The dynamic programming approach has been made more efficient over the years. The best known worst-case for the standard dynamic programming approach is $O(kn)$, achieved by a number of different algorithms and authors [35]. The best known average-case for this approach, developed by Chang and Lampe [8] is

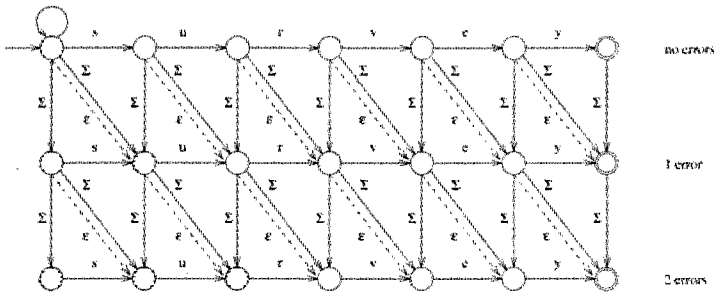


Figure 2: [35]A non-deterministic finite automaton (NFA) for matching the pattern 'survey' allowing two edit operations.

$O(kn/\alpha)$, where α is the size of the alphabet considered. For example, over case-insensitive English text, α is twenty-six.

2.1.2 Finite Automata

Also a fairly old approach to approximate string matching, the basic idea behind string matching with finite automata is that we can construct a finite automaton out of our pattern and feed the text through the automaton, one character at a time. Whenever a final state is active, we have found an approximate match. Consider the non-deterministic automaton displayed in Figure 2 [35]. Each row of the automaton denotes the number of errors encountered so far. Every column denotes the matching of a prefix of the pattern. Thus, if a state in column m and row i is active, it means we have approximately matched the pattern to the text with i errors. The transitions of the automaton are described as follows. A horizontal line represents a character match. A vertical line represents an insertion in the pattern. A solid diagonal line represents a substitution in the pattern, while a dashed diagonal line (with an ε-move) represents a deletion in the pattern. A practical implementation of this automaton is not intuitive. Earlier in the development of finite automaton approximate matching systems, a *deterministic* automaton approach was

attempted where each possible transition in the dynamic programming matrix became a state in the DFA. While easy to visualize in concept, it is impractical. Going from one cell to another in the dynamic programming matrix can occur three different ways. We can move horizontally, vertically, or diagonally. This means we must make allowance for 3^m different states if we wanted to ensure every combination of transitions were available. This approach is unfeasible for any practical pattern length, as 3^m quickly explodes. It was not until the development of bit-parallelism that this approach was able to take off using *non*-deterministic automata.

2.1.3 Bit-Parallelism

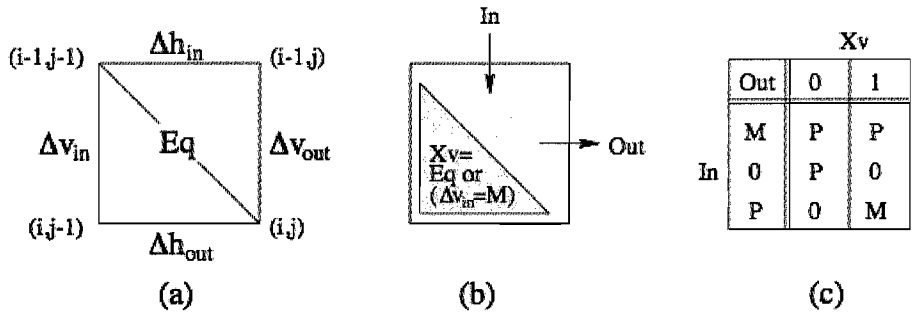
The main idea behind bit-parallelism is that we can use the bits in a single computer word to represent more than one entity. This saves on space and allows us to cycle through iterations of a program by using logical bit operations (such as shifts, ORs and ANDs). Since logical bit operations are fast and operate on every bit in a computer word, we can update w states or entities of an algorithm in one quick operation, where w is equal to the number of bits inside a computer word. Computer words typically consist of either 32 or 64 bits.

Finite Automata and Bit-Parallelism We will consider the implications by revisiting the finite automaton solution from a bit-parallel point of view. Imagine the length of our pattern was less than or equal to the number of bits in a computer word. A state in an NFA is either on or off, much like a computer bit. This means we can represent the entire automaton in Figure 2 in k computer words, one word for each row of the automaton. We can also update the entire automaton in just k bit-wise operations, one operation for each row of the automaton. This would lead to a very efficient $O(kn)$ worst-case complexity. If we do not know the length of our pattern ahead of time, a more general worst-case com-

plexity is $O(kn\lceil m/w \rceil)$. In 1992, Wu and Manber [45] found a practical way to implement the NFA using the bit-parallelism described above. They started by building a table B, where each entry pertains to a letter of the alphabet. For each character c, $B[c] = b_m...b_1$. Bit b_i is set to 1 if the i^{th} character in p is equal to c. For instance, if our pattern is *progress*, $B[r] = 00010010$ and $B[s] = 11000000$. They then developed an algorithm that fits each row of an automaton like that in Figure 2 into a separate computer word, defined as R_i. As we are allowed up to k errors, the algorithm uses $k + 1$ computer words for the automaton. For each text character encountered, the transitions of the automaton are simulated using logical bit operations on the computer words. The formula to obtain the new R values, R', at text position j using the old R values is as follows.

$$\begin{aligned} R'_0 &= ((R_0 << 1)|0^{m-1}1)\&B[T_j] \\ R'_{i+1} &= ((R_{i+1} << 1)\&B[T_j])|R_i|(R_i << 1)|(R'_i << 1) \end{aligned}$$

At each text character, R_0 (which is initialized to an all-zero vector) is shifted to the left and the empty right-most bit is filled with a 1. A logical AND is then performed between the resultant vector and $B[c]$, where c is the text character just read. This represents the first row of the automaton, as a bit in this vector will be set to 1 if and only if that state has been reached without an error thus far. The second row of the formula represents the $(i + 1)^{th}$ row of the automaton, which has a state active if and only if the state can be reached with $(i + 1)$ errors. The $(R_{i+1} << 1)\&B[T_j])$ represents a character match from a previous state in the same error-row. The $|R_i|$ represents an insertion from a row with one less error. The $|(R_i << 1)|$ represents a substitution from a row with one less error, and the $|(R'_i << 1)|$ represents a deletion from a row with one less error. If $R_i[1] = 1$, we know we have an approximate match with i errors. This is an efficient algorithm, as it reaches the $O(kn\lceil m/w \rceil)$ complexity mentioned earlier.



Process from horizontal and vertical cell differences to output enumeration. Myers used the variable X_v to enumerate all 18 possible combinations into the simple table seen in (c). Eq is set equal to $B[T_j]$ as described in the previous section.

Figure 3: Cell Input/Output for Myers' Bit-Parallel Algorithm

Dynamic Programming and Bit-Parallelism We will revisit the dynamic programming approach by looking at a solution that uses bit-parallelism by Myers in 1999 [34]. Myers realized that the difference from one cell to the next in the dynamic programming matrix is never greater than one. Myers then developed a formula that could determine the value for a current cell given the horizontal and vertical differences of the surrounding cells (to the top and left) and the match value between the text and pattern characters. Given that there are only three possible difference values (-1, 0, and 1) for the horizontal and vertical transitions, and that there are only two possible match values (match and no-match), there are only 18 possible combinations of difference and match values. Myers enumerated all 18 possibilities and found the most efficient formula possible (using only logical operators) to express the value of the current cell given the three aforementioned input values. As each text character is fed through, the entire dynamic programming matrix is updated in $\lceil m/w \rceil$ iterations. This allows us to report an approximate match if any of the cells in the bottom row contains a value less than or equal to k. The resultant algorithm is very fast, achieving a worst case of $O(n\lceil m/w \rceil)$. It is clear that when the pattern is shorter

than the size of a computer word, the algorithm achieves the theoretical best worst-case for approximate string matching of $\Omega(n)$. Myers' algorithm is the fastest approximate string matching algorithm of its class. However, all the algorithms looked at thus far did nothing to preprocess the text. The algorithms looked at thus far simply fed the text through the algorithm and gained an output. There are other classes of approximate string matching algorithms that perform much faster in practice, but they process the text first, using the information gained to increase the performance of the matching algorithm.

2.2 String Indexing

String indexing methods are a relatively new approach to approximate string matching. They are useful when we have an extremely large text that is searched frequently with many patterns. When aligning millions of oligonucleotides to a reference sequence, the benefits of string indexing are obvious. Many indexing methods have been developed for exact string matching (in other words, for $k = 0$), but it is only recently that indexes have been modified to accommodate approximate matching. Many indexing methods require a traditional algorithm to verify matches once a set of candidate matches has been found. Let us define a simple index structure that can be assumed for the remainder of this section [37]. We pre-scan the text or pattern and make an archive of all occurrences of each substring of a specific length (called the *query size*). We archive these pattern occurrences by converting each character in the pattern to a number and inserting the number in ascending order along with its position in the text. For example, take the text $AAAACCGAAAAG$ with a query size of four. The first pattern considered ($AAAA$) will be converted to the number 1111 at position one. The next pattern ($AAAC$) will be converted to the number 1112 at position two. After each length four substring has been converted and sorted, we will be left with a sorted array with the first few values given below:

Index	Value	Position
1	1111	1
2	1111	8
3	1112	2
4	1113	9

We will also have an accompanying data set that contains the start and end indices for each pattern value. For example, the first three lines of our data set will look like:

Pattern	Start Index	End Index
1111	1	2
1112	3	3
1113	4	4

Consider trying to find all occurrences of *1111* within our text. We start by looking up *1111* in our second data set. It tells us that *1111* exists between indices one and two in the first data set. Looking up indices one and two in the first data set tells us that *1111* exists in positions one and eight within our text. This is a very fast process, and when one expands this example to a much larger text, the advantages are obvious. There are some considerations involved when creating an index. For instance, imagine your pattern length is 36, and you are searching across a DNA text. This means that there will be 4^{36} possible entries in your index if you want the query size to equal your pattern length. An index of this size is simply impractical, so we must choose a smaller query size. A second consideration is the fact that we are performing approximate string matching, and so we do not always want to be searching for an exact match on our index. Three approaches to these considerations are detailed below. After these three approaches, a more modern, popular approach will be detailed in its own section, known as the Burrows-Wheeler transform. This approach has very recently gained much popularity in the way of string matching for computational biology.

2.2.1 The Word Neighbourhood

Navarro described a method for approximate string matching using an index called the *word neighbourhood* [37]. The basic idea behind a word neighbourhood for a pattern p and an edit distance k is that the neighbourhood will contain all words that are within k edit operations of p. Thus, if we can generate the k-neighbourhood for p, we can simply search every word in the neighbourhood on the index and record every hit as an approximate match at that location within the text. This would initially sound like an extremely simple and easy solution. However, the size of the neighbourhood can quickly explode. The size of a word neighbourhood has been bounded at $O(m^k\alpha^k)$ [43]. As an example, if we were to match length 36 DNA reads allowing up to two errors, we would have to search a word neighbourhood of 20 736 words in order to find all approximate matches for one read. If we were to allow one more error, that number becomes 2 985 984. Since the size of the word neighbourhood increases so rapidly, it is only practical for extremely small m and k.

2.2.2 Exact Partitioning

Navarro described another method for approximate string matching over an index [37]. It is easy to see that in every approximate match of a pattern p, there are sections of p that match the text exactly. If we allow k errors in an approximate match, then if we were to split out pattern into $k+1$ sections, one of the sections is guaranteed to match exactly by the pigeonhole principle as we cannot fit k errors into $k+1$ sections. Imagine we build an index with a query size of $\lceil m/(k+1) \rceil$. We can split each pattern we are to match into $k+1$ sections of length $\lceil m/(k+1) \rceil$. We can search each section over the index, and for each hit we encounter, we verify the surrounding text for a potential match with an inline approximate string matching algorithm. Exact partitioning is a useful method when we have a moderately sized k. If k is too small, our query size might be too large, and our

resulting index may take up too much memory to be practical. If k is too large, our query size will be so short that we will get too many false positive hits and we will be verifying too many false-positives to make it an effective solution. One other problem is present. When $k+1$ does not divide evenly into m, we will have an overlap in some of our sections. For instance, if our pattern is $AAACCCGGGT$ ($m = 10$), and we are allowing two errors ($k = 2$), our query size is going to be four. Since three sections of four characters equals twelve characters, some of the characters in our pattern will be repeated across multiple sections. If one of these characters happens to be the location of an error in the pattern, the exact partitioning method might not pick up an approximate match even though one exists, as one error will be spread across multiple sections.

2.2.3 Intermediate Partitioning

The most recent of the three approaches is intermediate partitioning, which was introduced by Navarro in 2000 [36]. It combines the best of the two previous approaches and produces better results. In 1994, Myers showed that the optimal query size for an index is equal to $\log_\alpha n$ [33]. An optimal query size means it will be short enough to allow a practical index size but long enough to avoid having too many false-positives. The intermediate partitioned approach by Navarro uses this optimal query size to build an index. We then split our pattern into $j = \lceil m/\log_\alpha n \rceil$ sections of length $\log_\alpha n$. Similar to the exact partitioned approach, one of the sections is guaranteed to have at most $\lfloor k/j \rfloor$ errors. Then, similar to the word neighbourhood approach, we generate the $\lfloor k/j \rfloor$-neighbourhood for each section. If any of the words in the neighbourhood returns a hit on the index, the surrounding text is then checked for an approximate match using a traditional approximate string matching algorithm. Intermediate partitioning has advantages over the word-neighbourhood and exact partitioning in that its query size is always optimal. This means we will check fewer

false-positives across a smaller index in general when compared to the other two methods. Intermediate partitioning has the same problem as exact partitioning, in that if our query size does not divide evenly into our pattern length, errors that occur in section overlaps can falsely increase the edit distance between the pattern and a text, leading to ignored hits.

2.2.4 Comparing the Three Approaches

I compared the intermediate and exact partitioned approaches in a practical way by programming each algorithm separately and racing them against each other. The word neighbourhood approach was not attempted because it is obviously not feasible for any length read generated by next-generation sequencing technologies. The tests were done by mapping 5000 length 36 Solexa reads to the first 63 000 000 base-pairs of chromosome nineteen. I compared the two approaches for $k = 2, 3, 4, 5$. The algorithms were implemented in their most basic form, using the C programming language. Myers' bit-parallel dynamic programming approach was used as the online algorithm for both approaches in order to verify potential matches. As can be seen in Table 1 and Sections 2.2.2 and 2.2.3, the exact partitioned approach begins to waste time verifying false-positives as k increases, due to the decreasing query size. Even when k is equal to two, the intermediate approach is superior due to the fact that it always uses an optimal query size. For $n = 63000000$, $m = 36$ and $k = 2$, the intermediate approach used an optimal query size of 13 whereas the exact partitioned approach used a query size of 12. However, it has recently been discovered that the Burrows-Wheeler transform performs exceptionally well in an approximate matching role and has become extremely prominent in NGS software due to its effectiveness [26, 30, 17].

Table 1: Comparison of the Intermediate and Exact Partitioned Indexing Approaches

	2 errors	3 errors	4 errors	5 errors
Exact Partitioning	60.521s	295.656s	2876.491s	9385.815s
Intermediate Partitioning	34.657s	34.796s	35.069s	35.775s

2.3 The Burrows-Wheeler Transform

In 1994, Burrows and Wheeler described a method to efficiently compress large texts [5]. Called the Burrows-Wheeler transform (BWT), its original usage was not intended for computational biology or even approximate string matching. When looking at the abstract of the initial publication, it is clear that the algorithm was only intended to be used as a good compression algorithm. However, recent publications [26, 30, 17] have shown that the Burrows-Wheeler transform is an excellent tool in an indexing scheme for approximate string matching in NGS platforms.

2.3.1 The Transformation

Although some NGS alignment platforms do not make use of the compressible qualities of the resultant transform[26], it is useful to know how the transform is created, as it is paramount in understanding how string matching over the BWT functions internally. Assume our text $t = GTCAGC$, and thus $n = 6$. The compression transformation is as follows. We create an n by n matrix, M, where each row in the matrix is a cyclic shift of t, sorted in lexicographical order. This guarantees that at least one of the rows of our matrix contains the original text, t. Along with M, we will have an index, I, that denotes the first row in the matrix that is equal to our original text. The matrix for our sample text is below.

row	pos. in text	rotation
0	3	AGCGTC
1	2	CAGCGT
2	5	CGTCAG
3	4	GCGTCA
4	0	GTCAGC
5	1	TCAGCG

This is our matrix, M, and it can be seen that our index, $I = 4$. The entire transform is simply equal to the last column of our matrix, paired with our index. The transform for our sample text is ($L = CTGACG$, $I = 4$). Given such a short input string, it is hard to see why this transform might lead to effective compression. However, in the context of the English language, it is easy to see. Consider a large English text. There will likely be many instances of the word 'the'. The results of the transform will group all of the rotations starting with 'he ' together, leading to many consecutive rows ending with the letter 't'. Similarly, when looking at consecutive rows in the matrix starting with the letter 'n', the ending characters of these rows are likely to be vowels, creating a much higher likelihood of equal characters occurring consecutively. We can then apply popular techniques such as move-to-front coding followed by Huffman encoding to create a compressed version of the text. These techniques will not be explained here as the compression properties of the transform are not utilized by BWA and as a result, compression is outside the scope of this work.

2.3.2 The Reverse Transformation

If we are going to transform a text using the Burrows-Wheeler transform, we are going to have to reverse the transformation to retrieve the original string. We begin the reverse transformation with only the knowledge of the last character in each rotation and I. However, it is easy to retrieve the first character in each rotation. Since the rotations are sorted lexicographically, it is obvious that the first column, F, of our matrix M can

be derived from L by sorting it lexicographically. Given $L = CTGACG$, it is clear that $F = ACCGGT$ by sorting L lexicographically. We now create an array, T. If $T[i] = j$, it implies that if $L[i]$ is the k^{th} occurrence of some character in L, then $F[j]$ is the k^{th} occurrence of that same character in F. Thus we can say that $F[T[i]] = L[i]$. In our example, $T = \{1, 5, 3, 0, 2, 4\}$. Now, since each row is a cyclic shift of t, we know that $L[i]$ directly precedes $F[i]$ in the text. If we substitute $i = T[j]$, we get $L[T[j]]$ directly precedes $F[T[j]]$ in the text. However, we have defined T such that $F[T[i]] = L[i]$. We can substitute this equality and see that $L[T[j]]$ directly precedes $L[j]$ in the text. Now, given $I = 4$, we know that the last character is $L[4]$. We know that $L[T[4]]$ is the second-last character, and that $L[T[T[4]]]$ is the third-last character, and so on. We can now simply follow the text backwards through T, outputting each character in reverse, retrieving the original text $GTCAGC$.

2.3.3 String Matching Using the Transform

Manzini and Ferragina describe a way to exactly match strings to a text that has been processed by the BWT [11]. They note that inside the matrix of the BWT for a text is contained the entire suffix array for that same text, within prefixes of each row of the matrix. A large advantage of being able to search across the BWT is that we obtain the speed of searching over a suffix array combined with the compressible qualities of the BWT. Before describing the process of searching for patterns across the Burrows-Wheeler transform, we must first define some concepts. They start by noting that given the matrix M, if i is the r^{th} row ending with a character c, and j is the r^{th} row beginning with c, then $L[i]$ represents the same character within the text as $F[j]$. They denote this process as *LF Mapping*. They also define an array, $C[1, 2, ..., \alpha]$ such that $C[c]$ denotes the number of occurrences of characters that are lexicographically-lower than c in the text. For example, if our text is

Algorithm BW-Search($P[1,p]$)

$c \quad = P[p];$
$i \quad = p;$
$sp \quad = C[c] + 1;$
$ep \quad = C[c+1];$
while $((sp \leq ep)$ **and** $(i \geq 2))$ {

$c \quad = P[i-1];$
$sp \quad = C[c] + Occ(c, sp-1) + 1;$
$ep \quad = C[c] + Occ(c, ep);$
$i \quad = i - 1;$
}
if$(ep < sp)$**return** 'not found' **else return** 'found ' + $(ep - sp + 1)$ + ' occurrences';

Figure 4: Algorithm for searching a pattern over a BWT-processed text [11]

acaaccg, $C[a] = 0, C[c] = 3$, and $C[g] = 6$. With this array, the search algorithm in Figure 4 can be defined. The algorithm assumes the presence of a routine (or pre-calculated array) $Occ(c,k)$, which returns the number of occurrences of character c in the first k characters of L, the Burrows-Wheeler transformed text. Essentially what Occ provides is the r value for *LF Mapping*. In the context of LF Mapping, $j = C[c] + Occ(c, k-1) + 1$ and $i = Occ(c, k-1)$, such that $L[i] = F[j]$. We add r to the first occurrence of c in our "suffix array" (which is contained in the prefixes of the BWT) in order to map the character returned by Occ to the row for which it is the first character. The resultant sums for sp and ep are the start and end indices within M of all row prefixes that match our current suffix, which is of length $p - i$. Once i has been reduced to 1 we are left with the start (sp) and end (ep) indices of the occurrences of our pattern p in prefixes of the suffix array, which are contained in M. Note that while this algorithm provides suffix array indices for

a matched pattern, it does not actually make use of a suffix array at all. It only requires the Burrows-Wheeler transformed text.

2.4 DNA Sequence Alignment

In the context of this work, DNA sequence alignment is the process of matching short DNA sequence reads (which are equivalent to patterns p), to the reference genome, which is equivalent to the large text t. The edit distance is needed in the alignment process to accommodate sequence errors and variations between reference and sequence samples, both of which are quite common.

2.4.1 Sanger Sequencing

One of the first widely-adopted sequencing technologies was published by Frederick Sanger in 1977 [39]. Dideoxynucleotides (ddNTPs, or more specifically, ddATP, ddCTP, ddGTP and ddTTP) are nucleotides lacking a 3'-hydroxyl (-OH) group. Due to this lacking, adding a ddNTP to the end of a growing nucleotide chain will halt the growth of the chain indefinitely. This process was coined as *chain termination*, and as a result, Sanger sequencing is now widely known as the *chain termination* method. In this method, ddNTPs of each nucleotide base (A, C, G, and T) are given a unique label using a fluorophore with a specific colour. Using a strand of DNA without its complement in a polymerization reaction will cause it to act as a template, facilitating the creation of its complement alongside it. Adding a ddNTP to the solution terminates the reaction at all its available positions, leading to DNA fragments randomly stopped at all positions. We then use capillary electrophoresis to separate the resultant sequences by length at a resolution of one nucleotide. We use the combination of length and coloured termination point to determine that the nucleotide identified by the terminating colour exists at position *length* in our

DNA strand. While Sanger sequencing is extremely accurate (99%) and provides a long read length ($600 - 1000bp$), it is an expensive and slow method.

2.4.2 Next Generation Sequencing

The characteristics of the newer sequencing technologies such as accuracy, length, and speed are quite different in comparison to Sanger sequencing, as described below. These platforms can sequence billions of base-pairs of DNA per single run of an instrument, at a small fraction of the cost [38]. They are fast and (relatively) inexpensive, at the cost of reduced accuracy and sequence length. A general overview on how these platforms work, with a slight lean towards a popular [32] NGS platform from Illumina/Solexa is as follows. An important first step for all NGS platforms is template preparation. In order to quickly sequence billions of base pairs, we must have stable and plentiful single-stranded DNA templates. Generally all NGS platform templates contain fragments or fragmented pairs of genomic DNA attached to a solid surface for support. Pairs of genomic DNA are simply fragments that originate at each end (the $5'$ and the $3'$ ends) of the strand. The sequences of these fragment ends are known as *paired-end reads.* They are useful because they originate from different ends of the same DNA fragment, providing context for alignment at the later stages of the DNA sequence alignment process. We can position many of these short templates onto the same support surface in order to facilitate billions of simultaneous reactions [32]. Since most technologies are yet unable to detect the results of a single fluorescent molecule, each template must be amplified, such that many identical templates reside in the same general position on the support surface, facilitating a more noticeable event for the instrument to recognize [32]. An example of this process can be seen in the Illumina/Solexa platform. Forward and reverse (one from each end) primers (starting points for DNA replication) are covalently bonded to a glass slide for support. A single-

stranded template is added to the primers, and replication starts with a forward primer and ends with a reverse primer. All forward and reverse primers near this template will replicate this process, leaving us with a cluster of identical templates. The Illumina/Solexa platform can facilitate up to 200 million separated template clusters, all with free ends with which to attach a primer to in order to initiate the sequencing reactions [32]. The second step in sequencing used by most NGS platforms is the sequencing or imaging phase. Similar to Sanger sequencing, the sequencing stage consists of adding dyed terminators to the templates and observing the fluorescence emitted. Where NGS differs, however, is that some NGS technologies use what are called *reversible terminators*. That is, terminators that can be removed, allowing addition of the next set of dyed-reversible terminators. This allows for a quick looping through the entire strand. After the fluorescence has been analysed, the dyes and terminating points are *cleaved* from the rest of the strand. The residual dye is then washed away, allowing the next set of nucleotides to be added.

2.4.3 NGS Platform Output

Once the machine has finished a run, the data generated has to be put into some sort of text format for file storage. The *FASTQ* format has been widely-adopted, in spite of a lack of formal specification and incompatible variants [9]. The great advantage of the *FASTQ* format is its simplicity. The basic structure of a *FASTQ* file is displayed in Figure 5. The title and description line can contain information such as the run and sequence ID numbers among other bits of information. The sequence line contains the raw nucleotide data, and the quality line contains a quality score for each nucleotide, expressed as an ASCII character. A quality score is a value denoting how certain we are that the nucleotide chosen was in fact the correct call. It follows then that the quality line must be the same length as the sequence line, one character representing a score for each nucleotide,

also represented by one character. Most of the variations pertaining to the *FASTQ* format have to do with varying ways of expressing the quality score. This four line sequence is repeated for each sequence generated. In the case of paired-end reading, there will be two *FASTQ* files, one file pertaining to each mate. For instance, the third read in the first file is mated with the third read in the second file. Figure 6 contains a sample of the first two sequences in an Illumina/Solexa generated *FASTQ* file. Lines one to four denote the first generated sequence, and lines five to eight denote the second generated sequence. Lines one and five are the identifying title lines for the each sequence. The @ symbol is an indicator that the current line is the title line. The first number, *0* in the first line and *1* in the second line denote the sequence number. The last number, *1* for both these sequences denote which part of a pair is being represented. In one file, this number will always be one, and in the pair's other file, the number will always be two. Following this line is the raw sequence data. The third line is the character + followed by an optional repeat of the title line. In this case, the title was elected to not be repeated. The last line is the quality score line. The n^{th} character on this line represents the quality score for the n^{th} character on the raw sequence line. The standard used for quality scoring is known as the *PHRED* quality scoring system [9]. The Sanger standard for converting a character to a *PHRED* score is to take its ASCII value and subtract 33. ASCII characters valued from 33 to 126 are allowed in the scoring system, meaning a nucleotide's *PHRED* quality score can range from 0 to 93. The *PHRED* score is logarithmic, meaning a score

Line 1:	*@title and optional description*
Line 2:	*sequence line*
Line 3:	*+optional repeat of title line*
Line 4:	*quality line*

Figure 5: General Structure of a *FASTQ* Formatted Sequence [9]

Line 1:	@0/1
Line 2:	*CAAAGAGTTCTCCTTCTGTGGGGTTAAGGTGGAAT*
Line 3:	+
Line 4:	<<<<<<<<<<<<<<<<<<; <<<:<<<<<<<<< 8;:;;
Line 5:	@1/1
Line 6:	*GAGCAATAAAACTTTATTGGAACACGGCTGCATTA*
Line 7:	+
Line 8:	<<<<<<<<<<<<<<<<<<: 7 <<<<<<<<<<<<:::<

Figure 6: First eight lines of an Illumina/Solexa generated *FASTQ* file

of ten means there is a ten percent possibility of error, while a score of twenty means there is a one percent possibility of error. The formula for determining error possibility is $P = 10^{-Q/10}$, where Q is the quality score [9]. In Figure 6, most of the nucleotides have a quality score represented by the < symbol. The < symbol corresponds to ASCII value 60, meaning the quality score represented by < is 27. Since $10^{-27/10} = 0.001995$, the < symbol represents a 0.1995 percent chance that the nucleotide chosen was the incorrect choice, giving us 99.8005 percent certainty in the nucleotide. It should be noted that the FASTQ format lacks standards and specifications. For instance, in some *PHRED* scoring systems the @ symbol can be used to denote a quality score. What happens, then when the first nucleotide in a FASTQ formatted sequence has a score that maps to the @ symbol? There is ambiguity between the start of a sequence (denoted by the @ symbol) and the start of the quality score line. Standards for digital sequence formatting are greatly needed, as lack of standards will only compound problems in the future.

2.4.4 De-novo sequencing versus re-sequencing

Once we have the sequences generated by an NGS platform, we need to align them in some meaningful way in order to extract meaningful data from the sequences. If we already have a reference sequence to align them to, the high-level description of the alignment process

is quite simple. For each sequence encountered, find a matching (or closest-to-matching) region within the reference sequence. This is known as *re-sequencing.* However, if our species has never been fully sequenced before, we have no reference with which to align our sequences. This case is known as *de-novo sequencing.* For a recently developed efficient de-novo assembler and its methods, see [46]. Re-sequencing is the process of aligning short reads generated by an NGS platform to a reference sequence. This process assumes that a reference sequence exists. The process of re-sequencing serves as the link between approximate string matching and DNA sequencing. We treat the alignment of each short read to the reference sequence as its own approximate string matching problem. Many programs exist to serve this purpose, each with its own strengths, weaknesses, and variations. A couple of the more recent and popular programs will be described, finishing with BWA within its own section.

2.4.5 NGS Alignment Tools

SOAP SOAP (short oligonucleotide alignment program) was developed by Li *et al.* in 2008 [29]. The motivation behind the development of SOAP was to address the massive number of short reads generated by NGS platforms. Up until the development of SOAP, available software was too inefficient to deal with the amount of reads generated by NGS platforms. Moreover, available algorithms were optimized for longer read lengths generated by Sanger sequencing and were unable to cope with the short length reads generated by NGS platforms [29]. SOAP allows approximate matching using a restricted version of edit distance. Substitutions (called mismatches, or single-nucleotide polymorphisms) are allowed, however only one insertion or deletion of length up to three base pairs is allowed per read, and if a deletion or insertion is used, no substitutions are allowed. SOAP is also

compatible with paired-end reads, meaning it can use the context provided by paired-end reads to provide a more accurate alignment. SOAP works by first indexing the reference sequence. Unlike the simplified index described in Section 2.2, index table entries do not always consist of subsequences of a certain length. Recently, consecutive sequences have given way to *spaced-seeding*. With seeding, instead of matching k consecutive characters, a match occurs if a read matches an index location of k non-consecutive ("spaced") characters. For example, we may use a spaced-seed of 10011100111001, where matches are required at all positions where a 1 is recorded. To compare to more simplified methods, such as the one described in Section 2.2, an older approach would always have a seed of k consecutive ones, without any zeroes in between. Spaced-seeding has been shown to reduce false-positive hits while not compromising sensitivity, which in effect speeds up the program without losing the accuracy of an alignment [4]. This may be a surprising (and unintuitive) result. However, it has been shown that for spaced seeds, the probability of an actual approximate match given a seed match is greater than for non-spaced seeds [4]. After SOAP has indexed the reference sequence in this manner, we can start the alignment process. For each read encountered, SOAP creates the same seeds as were used to build the index and searches the corresponding index for candidate matches. The candidate matches are verified or tossed out and the results are reported. In 2009, SOAP was succeeded by SOAP2, which uses the Burrows-Wheeler transform to perform alignment. It is 20 to 30 times faster than the original SOAP, and has only one-third of the previous memory requirement [30]. Since an extensive description of how BWA uses the Burrows-Wheeler transform to facilitate DNA sequence alignment will be provided, a similar description is skipped here.

MAQ The predecessor to BWA, MAQ was developed by Heng Li in 2008 [28]. It differs from SOAP in that only substitutions (mismatches) are allowed in MAQ's error function.

Insertions and deletions (gaps) only come into play when a read has too many mismatches but its mate-pair (the other end of the paired-end read) has already been mapped. At this point, a gapped alignment is attempted, using the paired-end information as context [28]. MAQ also introduced the concept of quality scoring for alignments. If a read maps to multiple locations, MAQ will calculate a quality score for each of the possible alignments, assigning the read to the location with the highest score. MAQ makes use of mate-pair information, the PHRED score contained in the FASTQ input files, and Bayesian statistical models in order to calculate the quality score. MAQ also differs from SOAP in that SOAP indexes the reference sequence, whereas MAQ indexes the reads. By default, MAQ builds six hash tables to ensure that a sequence with up to two mismatches (substitutions) will be hit. The six hash tables correspond to six different spaced-seeds. For each seed, MAQ takes the nucleotides at the "one" positions of the seed and hashes it into an integer. Reads that hash to the same integer are grouped together in the table. The integers in the table are then sorted [28]. To complete the alignment process, MAQ scans the reference sequence, one base at a time, applying each seed to the current position in the reference sequence and hashing the result into an integer. The resultant integers are then looked up on the computed hash table and candidate matches are found. The quality scores for candidate matches are then calculated and the results are reported. In 2009, MAQ was succeeded by BWA [26], which also uses the Burrows-Wheeler transform for alignment, and will be described later.

BWA BWA is the successor to MAQ, developed by Heng Li in 2009 [26]. Its aim was to improve MAQ by allowing gapped alignment while also giving a significant speed increase for large genomes such as the human genome [26]. It was written in the C programming language, and is available as a command-line tool that can be run on a standard desktop computer due to its moderate memory requirement (approximately 3GB for the human

genome). While having multiple functions, BWA is mainly used for short read alignment using the Burrows-Wheeler transform, so the concepts behind this function will be the main focus of this section. Short read alignment using the BWT is broken down into three main components, each with its own BWA command. Indexing the reference sequence *(index)*, determining the suffix array intervals for each DNA sequence by scanning the indexed reference sequence *(aln)*, and determining the best alignment for each DNA sequence by choosing the best alignment from the previously generated suffix array intervals *(samse/sampe)*. BWA was chosen as the focal point of this work because it is open source, supports input from many NGS platforms, and it is very fast.

2.5 Parallel Computing

Until recently, most software and hardware were designed to execute instructions in sequence. Sequential instruction execution means that only one operation can be executed at any given time, hence the many instructions in a complex algorithm are executed *in sequence*. Sequential computers are quickly reaching limits on how fast they can execute instructions sequentially [1]. The idea behind parallel computing is that we can have several processors communicating in a way such that they can simultaneously execute the same algorithm, as each processor can individually execute instructions at the same time. This results in an algorithm completing in a fraction of the time it would take on a sequential computer. While this is not a new idea in theory, recent reductions in the cost of processors has led to a feasible practical application of parallel computing. Computers, both sequential and parallel have been grouped into four classifications, namely SISD, SIMD, MISD and MIMD [44]. SISD stands for single-instruction stream single-data stream, which represents a traditional computer. These types of computers take one set of data and operate on the data using a single set of instructions. SIMD stands for single-instruction

stream multiple-data stream, which represents a simple parallelized algorithm running on a parallel computer or cluster. A data set (or multiple data sets) are distributed across multiple processors, and each processor executes the same instruction set on said data. This is the most common approach to parallel computing and can be assumed for the rest of the discussion. The other two classifications utilize multiple instruction sets and will not be considered as they are rare and not related to this work. Some practical approaches to SIMD parallel computing will be detailed below.

2.5.1 Multithreading

In modern operating systems, a process is an entity used to "group resources together" [42]. Traditionally, a process had one thread of control, meaning only one instance of program execution can exist for the duration of the process. Once the control thread was finished execution, the process would end. In more modern systems, a process can have multiple threads of control, where each thread's current state of execution is independent from all other threads, but each thread shares resources and global variables [42]. On single processor systems, multiple threads ("multithreads") provide the illusion of parallelism, as the processor will quickly switch back and forth between each thread, making it appear like they are being executed simultaneously. True parallelism fails here, as the task will still take as long to complete as if there were only one thread. However, on multi-processor systems, we can assign each processor a thread, and something close to true parallelism can take place [42]. It is not truly parallel, as the threads share the same process resources. True parallelism involves running multiple *processes*, not threads, in parallel. However, multithreading on a multiple-processor system can greatly increase the speed of a running process and since memory is shared, a multithreaded application can require less RAM than a parallel application. While shared resources can reduce on RAM, it can become

a major difficulty when attempting to write an efficient multi-threaded program for use on a multi-processor system. If all threads are attempting to modify a global variable, one thread will block all other threads out, requiring them to wait. Shared resources can severely degrade the efficiency of a multi-threaded application unless lock-free operations are used.

2.5.2 Parallel Computers

A parallel computer is either a single computer with multiple processors or multiple computers interconnected to form a high-performance system [44]. In general, there are two types of parallel computer. The first, which was not considered for a parallel version of BWA is a *shared-memory multiprocessor system* (SMP). These systems consist of multiple processors, each interconnected to the same physical RAM. A benefit of this type of parallel computer is that each processor shares memory and can thus share global variables, much like a multithreaded application. A problem with SMPs is that due to the fact that each processor needs an interconnect into each block of RAM, it is difficult to create SMPs with large numbers of processors [44]. The second type of parallel computer is a *message-passing multicomputer*. In this model, each computer (or *node*) has its own local RAM that cannot be accessed by other nodes. Each node can communicate by sending messages (data) to each other via a message passing interface [44] over some sort of interconnection network. This model scales well, in that we are able to add many nodes to a multicomputer, allowing for many processors to execute an algorithm in parallel. There are many different ways of interconnecting nodes, such as mesh, hypercube and cluster [44]. Their construction are as follows.

Mesh A square mesh can be constructed by connecting each node to its nearest four neighbours, like a two-dimensional grid. Edge nodes are connected to their nearest three

neighbours, and corner nodes are connected to their nearest two neighbours. In the worst case, passing a message through a mesh with p nodes takes $2(\sqrt{p}-1)$ communications, if, for example, the message originates at the upper-left-most node and is destined for the lower-right-most node [44].

Hypercube In general, an n-dimensional hypercube consists of 2^n processors. Two nodes, u and v are connected if the binary representations of their "node numbers" differ in only one position. In the worst case, passing a message through a hypercube with p nodes takes $\log p$ communications [44].

Cluster The problem with the above solutions is that they are specially-designed systems and so they are expensive. While these systems scale fairly well, in a few years, newer, faster processors will render an older parallel computer obsolete. Cluster computing solves all of these problems, and as a result has become the most cost-effective, widely-used message-passing multicomputer in use today [44]. A computer cluster consists of many regular workstations networked together. This is effective because regular workstations are available at low cost. Additionally, workstations can be easily added to a cluster to make it larger and do not even require the same specifications as the rest of the cluster's workstations. An added bonus is that with a cluster of workstations, we can run existing sequentially-written software on it, or simply modify it to run in parallel using a message-passing interface (MPI) [44]. Each individual workstation is commonly networked via ethernet cables and is addressed with TCP/IP as it would be on a typical network [44].

2.5.3 Parallel Algorithms

Speedup It is clear that parallel algorithms provide potential for speedup. Speedup for a parallel algorithm is defined as a measure of performance relative to its sequential

performance, and assumes the sequential algorithm is in its most optimal form.

$S(p) = \frac{t_s}{t_p}$ [44], where t_s is the execution time for the sequential algorithm, and t_p is the execution time for the parallel algorithm. Since we are assuming t_s is optimal, the quickest execution time for a parallel algorithm using p processors is bounded as $\frac{t_s}{p}$. Thus, our maximum achievable speedup is $S(p) = \frac{t_s}{\frac{t_s}{p}} = p$.

Efficiency It is useful to know how much each processor is being utilized in a parallel algorithm. There will be times where some processors are idling while others are working. We define efficiency as $E(p) = \frac{t_s}{t_p \times p} \times 100\%$ [44]. When we substitute in our formula for speedup, we get $E(p) = \frac{S(p)}{p} \times 100\%$. Since we have shown that $S(p)$ has an upper bound of p, it follows that $E(p)$ has an upper bound of 100%. That is, no parallel algorithm can be more efficient than its sequential counterpart, assuming the sequential algorithm is optimal.

Scalability An extremely important property of a parallel algorithm is how well it scales. Scalability is the ability for a parallel algorithm to retain its efficiency as more and more processors are used to run the algorithm. The more scalable a parallel algorithm is, the more processors it can be run with, and the more benefit can be wrought from it. In most parallel algorithms, there are sections of code that are only executed with one processor. An example of this is an initialization step, where a master processor divides data between the remaining processors. Amdahl developed an equation that calculates speedup based on a ratio of code executed by a single processor to concurrently executed code [44]. The time taken to perform such a parallel algorithm with the ratio f is $ft_s + \frac{(1-f)t_s}{p}$. Speedup can be redefined then, as: [44]

$$S(p) = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}},$$

which can be reduced to:

$$S(p) = \frac{p}{1 + (p-1)f}$$

It is clear that as p approaches infinity, speedup becomes equal to $\frac{1}{f}$. Thus if 5% of our algorithm is spent with only one processor executing code, Amdahl states our maximum speedup will be 20, no matter how many processors we use. Essentially, Amdahl's law states that the larger the ratio of parallel to sequential execution time is, the more efficient our parallel algorithm will be. While this is a good approach to scalability, it assumes that as we add more processors to the system, our ratio f will stay constant. Gustafson proposed that as a parallel system increases in size, the size of input data that is able to be processed in a reasonable amount of time also increases, and proposed that this fact must be taken into account. Gustafson did this by assuming the parallel execution time stays constant. In other words, if we double p, we double our input data to maintain a constant t_p. Gustafson expressed his scaled speedup as follows: [44]

$$S(p) = p + (1-p)ft_s$$

Gustafson states that scaled speedup is a line of negative slope instead of the rapid convergence proposed by Amdahl. The slope is negative because it is a *scaled* speedup factor, and thus we increase our input data as we increase our system size to maintain a constant t_p. Gustafson also showed speedup factors that had been practically achieved that matched

his law better than Amdahl's [44]. The two laws are not competitors, they are two different ways of describing scalability in parallel systems given different variables.

3 Parallelizing BWA

Prior to actually going ahead with the parallelization of a sequential algorithm, it is necessary to understand how the algorithm works. An in-depth description of how BWA works internally will be given as a groundwork for the parallelization process.

3.1 BWA

A more in-depth look at BWA than the description given in Section 2.4.5 will be given below, detailing the behind-the-scenes methods BWA uses to align short DNA sequence reads using the BWT.

3.1.1 Improving Memory Efficiency

We can use the Burrows-Wheeler transform to approximately match DNA reads onto a reference sequence very efficiently. As seen in Section 2.3.1, generating the BWT requires use of a matrix which is quadratic ($O(n^2)$) in time and space. When dealing with large values of n such as with the human genome, generation and storage of such a matrix is not feasible. However, it is shown in [16] that we can generate the Burrows-Wheeler transform from a compressed suffix array in $O(n)$ time and $O(n \log \alpha)$ space. It should be noted that generating the compressed suffix array from a text takes $O(n \log n)$ time. Before this algorithm can be described, we must first describe compressed suffix arrays.

Suffix Arrays Before we get started on suffix arrays we must introduce a new symbol, \$, which does not exist in any alphabet and is lexicographically smaller than all other characters in any alphabet. This character is placed at the end of our text, t. We assume that t is stored in an array, $T[0, 1, ..., n-1]$, where $T[n-1] = \$$. We define the suffix of T, T_i as $T[i..n-1]$. That is, all of the characters starting from $T[i]$ until the end of our text.

A suffix array of T, defined as $SA[0, 1, ..., n-1]$, is a sorted sequence of all the suffixes of T. $SA[i]$ denotes the starting position of the $(i+1)^{th}$ lexicographically-smallest suffix of T [16]. As an obvious example $SA[0] = (n-1)$ and $T[SA[0]] = \$$ for all texts. We also define $SA^{-1}[i] = j$, where $SA[j] = i$. $SA^{-1}[i]$ denotes how many suffixes are lexicographically smaller than T_i. An advantage of suffix arrays is that any pattern that matches the text at any point will occur as a prefix within at least one suffix of the suffix array. These prefixes are grouped together in contiguous order, since the suffixes are sorted lexicographically. This is the advantage of suffix arrays, that once we have found one matching prefix, every other matching prefix can be accessed in constant time. This makes searching over suffix arrays very efficient.

Compressible Suffix Arrays A simple form of the compressible suffix array of T is an array, $\Psi[0..n\text{-}1]$. The array is defined as follows. $\Psi[0] = SA^{-1}[0]$. For all other $i = 1, 2, ..., n-1$, $\Psi[i] = SA^{-1}[SA[i]+1]$. A naive approach to storing this array would take $O(n \log n)$ space ($\log n$ bits for every value). However, it can be seen in our example, that if $i < j$ and $T[SA[i]] = T[SA[j]]$, then $\Psi[i] < \Psi[j]$. In other words, if two suffixes, i and j, have the same first character, and suffix i is lexicographically smaller than j, then $\Psi[i] < \Psi[j]$. This is proven by Hon et al. in [16]. This means that the compressible suffix array consists of α sequences of increasing numbers. It can be seen in Figure 7 (c) that the letters $T[SA[i]]$ are grouped together and in lexicographically increasing order, and that within each group, $\Psi[i]$ is also increasing. Hon et al. used this fact to store the compressible suffix array using $O(n(H_0+1))$ bits in $O(n \log n)$ time, where H_0 is the order-0 entropy of the text. The Burrows-Wheeler transform, W, can be generated from Ψ using a simple formula defined in [16]. The formula is as follows. $W[\Psi^k[p]] = T[k-1]$, where $p = \Psi[0]$. $\Psi^1[p]$ means $\Psi[p]$, $\Psi^2[p]$ means $\Psi[\Psi[p]]$, and so on. Using this formula

i	T[i]	T_i
0	*a*	*acaaccg$*
1	*c*	*caaccg$*
2	*a*	*aaccg$*
3	*a*	*accg$*
4	*c*	*ccg$*
5	*c*	*cg$*
6	*g*	*g$*
7	*$*	*$*

(a) Suffixes of T

i	SA[i]	$SA^{-1}[i]$	$T_{SA[i]}$
0	7	2	*$*
1	2	4	*aaccg$*
2	0	1	*acaaccg$*
3	3	3	*accg$*
4	1	5	*caaccg$*
5	4	6	*ccg$*
6	5	7	*cg$*
7	6	0	*g$*

(b) Suffix Array of T

i	$\Psi[i]$	T[SA[i]]
0	2	*$*
1	3	*a*
2	4	*a*
3	5	*a*
4	1	*c*
5	6	*c*
6	7	*c*
7	0	*g*

(c) Compressible SA

Figure 7: The Suffix Array and Compressed SA of *acaaccg$* [16]

and given the Ψ array generated in $O(n \log n)$ time and $O(n(H_0 + 1))$ space as shown in [16], we can generate the Burrows-Wheeler transform for t in $O(n)$ time and space.

3.1.2 Indexing the Reference Sequence

Unlike MAQ, which indexes the input reads, BWA indexes the reference sequence by performing the BWT on it. This means the first step in aligning reads to any reference sequence is to index the reference. This function is aptly named the *index* function. It must be noted that the search method BWA uses requires a BWT to be built on the reverse reference sequence during this command as we need to search over both of the strands that make up a DNA double-helix. This command only needs to be performed once for each reference sequence we wish to align reads to, as the index is saved as a file that can be permanently stored on a hard disk. Thus if we have billions of DNA sequences taken from the human genome, we only need to index the reference sequence for the human genome once before any of the reads have been aligned. After the index has been built, we do not need to execute this command again.

Compressing the Reference Sequence The first thing done in order to build our BWT index is to compress the input file containing our reference sequence. BWA takes *FASTA* formatted reference sequences, originally in character form. Since the DNA al-

phabet is only of size four (A, C, G, and T), we can pack each character into a two-bit sequence. For instance, A can be represented by *00*, C by *01*, G by *10*, and T by *11*. If BWA encounters an N symbol (the nucleotide was non-determined by the NGS platform), it chooses a random base and inserts it, while noting the position of the N in a separate data structure. The compressed reference sequence is saved into its own file. This new file is then duplicated and the duplicate file reversed, as we are required to create the reverse BWT as well.

Calculating the BWT BWA computes the compressed suffix array as described in Section 3.1.1. From the compressed suffix array, it generates the BWT by using the formula given in Section 3.1.1. The process described in said section is the same as the process used in BWA, so no further explanation is needed.

3.1.3 Calculating the Suffix Array Intervals

The following will be a brief reiteration of the *BW-Search* algorithm shown in Figure 4, but explained in a way that matches the description given in [26]. Recall that the suffix array S for a string t is a permutation of the integers 0 through $n-1$ such that $S[i]$ is the start position of the i^{th} lexicographically smallest suffix in t. We can now define a suffix array interval, which is a pair $(R_s(W), R_e(W))$. $R_s(W)$ is the minimum index such that W is a prefix of $t_{S[s]}$, and $R_e(W)$ is the maximum index such that W is a prefix of $t_{S[e]}$. This interval corresponds to the *sp* and *ep* variables defined in Figure 4. If we have the suffix array interval for a pattern p, we can easily determine each exact occurrence of p in t by retrieving the value given by $S[k]$, where k references each value in our suffix array interval.

For exact string matching, we will have at most one suffix array interval per pattern. For approximate string matching, we might have many intervals per pattern.

For Exact Matching Determining the suffix array intervals for an exact match is done as follows. Let $C[c]$ be the number of symbols in t (not including \$) that are lexicographically smaller than c. Let $Occ(c, i)$ be the number of symbols in the first i characters of our BWT compressed version of t that are lexicographically smaller than c. We begin by setting our current character, c, equal to $p[m-1]$. Thus our beginning suffix array interval is $(C[c], C[c+1]-1)$. We work backwards through our pattern, and for every character, a, encountered in reverse order, $R_s(aW) = C(a) + Occ(a, R_s(W)-1)+1$, and $R_e(aW) = C(a) + Occ(a, R_e(W))$. Once we have worked to the front of our pattern, if $R_s(P) < R_e(P)$, p exists in t in all locations referenced by our suffix array interval. This method is known as backward search [26].

For Approximate Matching BWA uses an extension of backward search in order to facilitate approximate matching for up to k differences. In its most basic form, BWA generates the word-neighbourhood for each pattern and runs the exact match algorithm on each word in the neighbourhood. BWA improves the efficiency of the word-neighbourhood approach by estimating the lower-bound of the number of mismatches of a prefix of p. Since the bound is based on a prefix of p and our efficient search algorithm runs backwards, we need to use the reverse of the sequence in order to estimate the bound. The algorithm in its most basic form is illustrated in Figure 8. The resultant D array can be used to crop out large amounts the word neighbourhood and make it very efficient. $D[i]$ is our estimated lower bound of the number of mismatches in $P[0, i]$. Thus if we are performing backward search on a word W in our word neighbourhood and $D[m-1]$ is already greater than k,

Algorithm CalculateD(P)

```
z = 0;
j = 0;
for i = 0 to m − 1 do
   if P[j, i] is not a substring of T then
         z = z + 1;
         j = i + 1;
   end if
   D[i] = z;
end for
```

Figure 8: Algorithm estimating the lower bound of the number of mismatches in a pattern's prefixes [26]

we can skip the word entirely. As we progress along backwards search we will be able to discard other words as well as errors in the pattern decrease our remaining k threshold. The backward search algorithm extended for approximate matching is displayed in Figure 9. It assumes the presence of the CalculateD algorithm displayed in Figure 8. For each pattern, our D array is calculated using CalculateD. We then enter our recursively defined algorithm, CalcSAIs. The parameters are as follows. P is our pattern, i is the index of the current pattern character in our backward search, k is the number of errors we are yet allowed to encounter, and q and r represent our suffix array interval. We start on line 1 by checking if the estimated lower bound for the number of errors in our pattern is greater than k. If it is, we already know there cannot be a match and we return nothing. We continue on line 2 by checking if our current character is less than 0. If so, it means we have reached the end of the backward search and we return our suffix array interval. Intuitively, this may not make sense to have on line 2, but remember that it is a recursive algorithm. Line 4 represents the first part of our approximate matching. We make a

Algorithm CalculateSAIs(P, k)

Calculate C and Occ arrays from BWT
Calculate Occ' array from reverse BWT
CalculateD(P); /*uses C and Occ' arrays*/

return CalcSAIs($P, m-1, k, 1, m-1$);

Function CalcSAIs(P, i, k, q, r) {

```
1.  if k < D(i) then return NULL;
2.  if i < 0 then return I;
3.  I = NULL;
4.  I = I∪ CalcSAIs(P, i − 1, k − 1, q, r);
5.  for each b ∈ [A, C, G, T] do
6.      q = C(b) + Occ(b, q − 1) + 1;
7.      r = C(b) + Occ(b, r);
8.      if q < r then
9.          I = I∪ CalcSAIs(P, i, k − 1, q, r);
10.         if b = P[i] then
11.             I = I∪ CalcSAIs(P, i − 1, k, q, r);
12.             else
13.             I = I∪ CalcSAIs(P, i − 1, k − 1, q, r);
14.         end if
15.     end if
16. end for
17. return I;
}
```

Figure 9: Algorithm for determining the suffix array intervals of a pattern against a BWT-compressed sequence. Note that this process is repeated for the reverse reference on a complemented sequence. [26]

recursive call to our function, lowering our backward search character position without searching and decreasing the number of errors allowed for this call of the function. This is the function call covering an insertion in the pattern, as we are skipping a character and moving on. Line 5 is the start of our word neighbourhood generation, as we cycle through every possible character at our current index. For each possible character b, we calculate the suffix array interval when adding b to the front of our currently-processed suffix. This happens on lines 6 and 7. We then test if our suffix array interval is still valid on line 8. If

it is not, we continue to the next iteration of the loop as our current character makes for an invalid alignment. If our suffix array interval is still valid, we can move on. Line 9 makes the recursive call assuming a deletion in the pattern, as we do not move backward from our current character but also decrease the number of errors encountered. Lines 10 through 13 handle substitution errors. If b matches the character in P at our current index i, there is no substitution error, so we simply make the recursive call moving on to character $i-1$ and keeping k the same. If b does not match, we make the recursive call to character $i-1$, but we also decrease k. After all the recursive calls have finished, we are left with a set containing all of the suffix array intervals representing indexes in our reference sequence that match our pattern p with at most k differences.

3.1.4 Determining the Alignments from the Suffix Array Intervals

Since approximate matching may generate multiple suffix array intervals, outputting the alignments given the intervals is not a trivial task. Being the successor to MAQ, BWA utilizes a similar strategy as MAQ when dealing with multiple candidate positions. BWA calculates the quality score of all possible matches, determining the best fit. The mapping quality score is based on a number of criteria, such as number of gap opens, gap extensions, and mismatches. Each type of edit operation has its own penalty weight in calculating a mapping score, although the weights are modifiable at run-time with command-line parameters [26]. The default penalty scores have gap opens as being by far the most detrimental to a mapping score, and mismatches the least detrimental. Unfortunately, unlike MAQ, BWA does not use the PHRED quality scores of individual bases for this stage. For paired-end reading, the alignment stage is more complex. Each sequence is first aligned in the same way they would be for a single-end alignment. After this, BWA uses statistical methods to estimate the maximum, average, and minimum insert sizes for the

entire group of sequences. These sizes represent the statistical low, average, and high gap sizes between one pair and its mate. BWA then attempts to use the single-end alignments and the insert size estimates to map one read to its mate. Mates that BWA fails to align and/or pair are aligned using the Smith-Waterman algorithm [41].

It is clear that the longer our reads are, the quicker this phase will be (assuming k is constant). If our reads are longer, there is a smaller chance that there will be multiple "good" alignments. For a four character alphabet, the number of combinations of characters in an m-length pattern is 4^m. It is obvious that the greater m is, the greater the number of possible combinations resulting in a much smaller chance (namely $1/4^m$) that a match can occur at random. This means that there will be a smaller number of suffix array intervals to go through and thus less time will be wasted discarding candidate hits. Conversely, if our reads are shorter, such as the 32bp Illumina/Solexa reads used in the work of this thesis, this phase will take longer as there will be a larger number of suffix array intervals to look up. After all of the sequences have had their alignments determined from their respective suffix array intervals, we have completed the alignment phase. Alignments are outputted in the SAM file format [27].

3.2 Why a Finely-Grained Approach is not Practical

There are two broadly described ways to parallelize an alignment program such as BWA. The first method is to distribute q reads among p processors and have each processor align q/p reads. There are many ways to accomplish this method, such as multithreading, sequential read distribution, coarse-grained parallel read distribution and embarrassingly parallel read distribution. The second method is to distribute the workload of each individual read among many processors. This would be classified as a fine-grained parallel approach, one that requires much communication between processors. This would work

as follows. A master processor would handle most of the initial work by itself, namely loading the index into memory, loading each read into memory, and setting up the alignment process. Then for each read to be aligned, the master processor would distribute the m characters of our pattern over p processors, where each processor aligns m/p characters to the index. Then, each processor can communicate potential alignments with each other and find a set of alignments that work for the pattern as a whole. There are a few problems that make this approach impractical. First, aligning m/p characters to the index will produce many more false positives than if we were to align all m characters at once. This will result in much time wasted on verifying potential candidates. The reason for this is because m/p characters will always be less than m characters. This leaves us with the same problem that arises when determining alignments from suffix array intervals. As reads get shorter, the chances of random false-positive matches get higher. The second issue with a fine-grained approach is that calculating the number of errors in each match becomes more difficult. If we are allowing two errors and are using three processors, it is not enough to simply find suffix array intervals that represent a contiguous subsequence from each processor. The reason behind this is that each suffix array interval could contain an error, and concatenating three error-filled suffix array intervals will leave us with an unacceptable error ratio. While this problem is not insurmountable, it introduces extra computational complexity and will further decrease the efficiency of the resultant parallel program. The last problem is amplified by the first two, and that is communication overhead. The main problem with parallel computing in general is that processor-to-processor communication is costly and is something that never occurs in a sequential program. This is why a parallel program can never be improved in terms of total processor cycles when comparing it to a sequential version. Finely-grained approaches such as the one above require more processor-to-processor communication than coarse-grained and embarrassingly

parallel programs. For these reasons a finely-grained approach was not a practical solution to attempt.

3.3 Practical Considerations

Before attempting to parallelize BWA there were a number of parameters that needed considering. They are listed below.

3.3.1 SHARCNET

The first parameter that needed addressing was obtaining the physical ability to compute in parallel. High-throughput parallel systems are hard to come by, and it is difficult to develop parallel programs on a standard home computer. I was able to obtain an account on SHARCNET. SHARCNET is a consortium of Canadian academic institutions (mostly from southern Ontario) who share a network of high performance computers [7]. It was developed to address a severe lack in freely available high performance computing in the country. Every SHARCNET user is granted equal access to resources and is permitted to run programs with up to 256 processors simultaneously.

Systems SHARCNET is comprised of many systems, all with varying specialities. One common theme with the main SHARCNET systems is that they are all *clusters*, as described in Section 2.5.2. Some clusters, such as *kraken*, offer a huge number of powerful processors (3760) but do not offer much in the way of efficient communication. These systems are best used for coarse-grained or even embarrassingly-parallel applications. Other systems, like *requin*, have much lower latency interconnects and are ideal for fine-grained applications as message passing will be much faster. Every user has a home drive that is accessible across all systems. Every user also has system-specific work and scratch drives that own a much higher capacity and are thus used for the actual running of the programs.

This makes sense as performing parallel I/O over a network drive would be extremely inefficient compared to parallel I/O performed on a local drive. Every user is given a priority that increases or decreases depending on the amount of time the user has been taking up on system resources over the past two months. This priority, along with number of available processors are the determining factors in how fast a submitted job will begin.

Inconsistencies SHARCNET has proven to be a very volatile group of clusters. Systems regularly go off-line and are inaccessible for days to up to weeks at a time [6]. A more serious inconsistency is the performance of I/O and processing across nodes within a system. For instance, running the same job on the same system with the same data twice in a row can produce very different results. This is due to the fact that you may not receive the same set of execution nodes across multiple runs of a program. As a result, one run can be given a "problem node" while the next will go without. A problem node can ruin results, as illustrated by the three test runs described below, all using the same parameters. The test runs attempted to align 50 million reads across 64 processors. The first test run resulted in 63 of the processors completing their alignments in about eight minutes with the final processor taking an additional twenty minutes to complete. The second test run was assigned a different set of nodes. Sixty-two processors completed their alignments in about eight minutes, while the final two processors took an additional twenty minutes to complete. The third test run was assigned a different set of nodes from the first two tests. In this test run, all processors completed their alignments in eight minutes. Certain sets of nodes can be specified when submitting a job, but this almost guarantees the job will sit in the queue for a prolonged period of time, bringing development to a stand-still. This volatility in the SHARCNET makes debugging and testing very difficult. It is difficult to know when a program is running slowly whether or not it is a bug within your code or a bad set of SHARCNET execution nodes.

3.3.2 MPI

Not only do we need a physical system that is able to run programs in parallel, we also need software compilers that are able to provide control to a parallel environment. BWA is written in standard C. Luckily, there exists a parallel library and compiler for C, called MPI (message passing interface) [12]. The OpenMPI library and compiler are provided on the SHARCNET. Simply adding an "include mpi.h" to the top of any source file you wish to execute in parallel is enough to gain access to the library, provided you call the parallel compiler instead of the standard compiler. What an MPI program does is run simultaneously on a set number of processors. Every processor runs the exact same code as every other processor. Within the program, we can determine the rank of any processor, and with the rank, we can control the flow of the program. For instance, if we wish to send a message to our numerical successor, we would assign our rank to the variable r and send the message to processor $(r + 1) \bmod p$, where p is the number of processors. We apply the modulus operation to ensure processor $p - 1$ does not attempt to send a message to processor p, which does not exist. Instead, $p - 1$ will wrap around and send its message to processor 0.

3.3.3 NGS Data Sources

Another parameter to consider was NGS data sources. Since there are many NGS platforms in use, the data sets used by Dr. Ping Liang's research group at Brock University was chosen, which are mainly comprised of Illumina/Solexa and ABI SOLiD sequences. The sequences from the Illumina/Solexa platform are 36 base pairs in length. Interestingly enough, because the read length is so short, the importance of an efficient suffix array interval to alignment process became much greater. SOLiD reads were used to a small

extent to ensure that my modifications were cross-platform compatible. The SOLiD reads used were 50 base pairs long.

3.3.4 BWA Parameters

BWA has many different options, many that will change the way the mapping quality score for a read alignment is calculated, and many that will change allowances in sizes and locations of insertions and deletions. Most if not all of these parameters do not affect the parallelization process, but they needed to be considered regardless. For my parallelization of BWA, the only parameter I specified was the number of errors allowed in each pattern. Since I used 36bp reads, I was recommended to use an error allowance of two. When using SOLiD reads, three mismatches were allowed and the colour-space parameter was also used.

3.3.5 Modifications to Standard BWA

Bug Fixes An additional difficulty arose due to the fact that BWA is still very much a work-in-progress. I started the parallelization process with BWA version 0.5.8c, which at the time, was the latest version available. During my work with BWA, another version was released (0.5.9) that was able to fix some stability issues, however there is still at least one non-fixed bug in version 0.5.9. This came to light when I first attempted to run BWA with SOLiD reads (up until February 2011 I was only running with Illumina reads). With my parallelization of version 0.5.8c, running the program with SOLiD reads would generate inexplicable segmentation faults. This was due to a bug in the software that causes alignments to generate segmentation faults when the first read in a block of reads is of zero length. Zero length reads appear to be non-existent with Illumina reads and rare with SOLiD reads, which is why the bug is so hard to replicate. BWA has a set of

data structures used to hold information about the current read being aligned. This data structure is given extra memory if the current read is longer than the then-current-largest read examined. However, the current-maximum is initialized to zero [21]. Thus if the first sequence encountered is of zero length, it does not exceed the current maximum and no memory is allocated [22]. This is what generates the segmentation fault. This error is fixed by initializing the current-maximum value to negative one instead of zero. As of February 26th, 2011, this fix has not been implemented in the current release of BWA. It should be noted that subsequent releases of BWA may provide additional stability to my parallelized version, provided they are added in manually.

Tweaks Another item of note was the estimation of insertion size for SOLiD reads during the alignment phase. BWA (both standard and parallel) employs statistical methods to estimate the maximum number of base pairs between an aligned read and the alignment of its mate-pair. We will be able to more quickly align an unmapped mate if this estimation is as tight as possible. For the Illumina/Solexa reads, the maximum size was very consistent (673-676bp for 100 million Solexa reads), and aligning unmapped mates was very quick. For the SOLiD reads, the maximum size ranged quite a bit (3600-7200bp) and as a result some processors were much slower at aligning unmapped mates than others. Occasionally, an extremely high value was encountered (137815, for example) and the corresponding processor would take hours to do what should take a couple seconds. Even though a specification of 5000 being the maximum insert size was made in the program parameters, BWA ignores this value if it can (in its own mind) correctly estimate another maximum value. Occasionally a value like the aforementioned was able to slip through the cracks of BWA's screening process and replace the defined 5000. The reason this maximum size is so important is because BWA uses Smith-Waterman alignment to align a mate that was unmapped by the Burrows-Wheeler transform. The Smith-Waterman alignment

algorithm is of complexity $O(nm)$, where n and m are the lengths of the two sequences being aligned. The length of the reference sequence in this case is determined by the estimated maximum insert size. To first address this issue, the *-A* parameter is used, which disables Smith-Waterman alignment of unmapped mates entirely. While this speeds up the alignment phase considerably, unmapped mates stay unmapped. Instead, an "override to the override" is proposed to disable Smith-Waterman for a set of reads only if the estimated maximum insert size is more than the value of the input *-a* parameter, which in our case was 5000. This will allow the insertion size estimator to tighten our bounds if they can be tightened, but also utilizes our defined maximum so we can avoid problems created by insert sizes such as 137815.

Improvements An improvement to the way BWA handles multithreading was also added into parallel BWA. In BWA version 0.5.9 (the current version as of the time this was written), the *aln* command supports multithreading. If BWA is being run on a *p-core* processor, then the *aln* command can be split into p threads and run at a higher efficiency. BWA handles multithreading as follows. Each thread cycles through every single sequence, locking each sequence as it is analysed. If the sequence being analysed is not currently assigned to a thread, the thread doing the analysis will "reserve" the sequence by setting its assigned thread value to its own ID value. The thread then has the leeway to go ahead and reserve $r-1$ more sequences, where r is some predefined constant (1024 as of version 0.5.9). This method of distributing reads for analysis taxes the efficiency of BWA. On a 24-core system, performing the *aln* command with 24 threads is 12 times faster than performing the command with one thread. An optimal threading would see performing the command with p threads be p times faster. Since parallel BWA can make use of multithreading, a modification was made to the way BWA handles read distribution for threads in order to improve the efficiency of threading. When this improvement was applied to BWA 0.5.9,

running the *aln* command with 24 threads was 16 times faster instead of 12. The way the improved distribution works is as follows. For each read, if the current value of the loop counter (mod) the number of threads is not equal to the current thread ID, the current thread skips the current loop iteration. This way no reads need to be assigned, and no reads need to be locked. The distribution is based purely on the loop counter.

3.4 Approaches

This section will detail the different attempts that have been made at parallelizing BWA. The first section has to do with the impracticality of generating the index in parallel, and the remaining three sections will detail the methods used in parallelizing BWA.

3.4.1 Generating the Index

Generating the index is mainly comprised of two sections: generating the Burrows-Wheeler transform and generating the suffix array. For each reference genome, the index only needs to be generated once. In fact, BWA's *index* command was run once at the beginning of this work and has not been run since. Since we only need to index one reference sequence for all human DNA reads, it was not practical to look into parallelizing the indexing process any further.

3.4.2 Reading the Index

Before we can perform the alignment, we must first obtain our input reads and reference genome. This is done by reading in the index created by BWA's *index* command from file. This is one of the areas that required the most thought and research in order to successfully parallelize.

BWA The sequential version of BWA only ever uses one processor to read in the index, even when using multithreading. The process behind reading the index into memory for sequential BWA is straightforward. BWA simply queries the size of the index file and reads in integers representing the index accordingly [23]. These are stored in a data structure that is utilized for the remainder of the program. In the case where the multithreaded *aln* command is being run, each thread is passed a pointer to the index data structure from the main thread [20]. This is an efficient method as it only takes a few seconds to read in and store in memory an index built for the human genome.

Parallelism without Communication Several methods for parallelizing the process that reads and stores the index into memory were tested. These methods were met with various levels of success. The first two attempts did not make any use of processor-to-processor communication, and are thus categorized as an embarrassingly-parallel approach. The first attempt was to leave the algorithm the way it was, and let all p processors compete for access to the index file. This approach worked fine for small genomes, but was not feasible for the human genome. When attempting this method with the human genome, some processors would get blocked out of the index file entirely by the rest of the processors. This is due to the fact that when reading in the index, one call of *fread* would request the entire index, instead of many smaller portions [23]. Thus there is no processing of a partial index that can be done while a processor is waiting for file access, resulting in wasted clock time. While most processors would fight for and receive occasional access to the index file and complete a read shortly, other processors would not receive access to the index file until the rest were finished, and would end up taking twice the usual amount of time to read and store the index. Since a run of a parallel program is only as fast as its slowest processor, it was concluded that this attempt was not efficient and a different strategy was needed. A second approach was taken that would ensure every processor would receive its

own index file. Since any genome only needs to be indexed once, it is not too arduous a task to duplicate the folder containing the index files p times, and assign each processor its own individual index folder and files as the duplication process would also only need to occur once (albeit p times). Using this method would ensure that there would be no competition for file accesses. Unfortunately, although no processors were locked out of their input files (every processor would immediately start to read), the strain put on the file server with even just twenty processors was too great and read times for the human genome were even worse than for the first approach. These two attempts show that communication among processors is needed when a large amount of parallel file I/O is required.

Parallelism with Communication A third approach was considered where one processor, designated as the master processor, would read and store the index, and then broadcast the data structure to all other processors. This has the advantage of only one processor accessing one index file, but has the disadvantage of all other processors idling while the first does its file accessing. The results were good. It takes a short amount of time for the master processor to read and store the index. Since broadcasting the data structure to all other processors is completed in $O(\log p)$ time, this is an efficient approach. This approach is most effective when used on a parallel system with efficient processor-to-processor interconnects. The *requin* system on SHARCNET can communicate data between processors at a rate of 800 MB/s [13], which is quite efficient when compared to the 100 MB/s we can expect from the file server [13]. This explains why the file system methods described above were not as efficient. A 800 MB/s rate means that when taking into account the forward and reverse strand indexes along with their corresponding suffix arrays, each iteration of the broadcast can theoretically occur in three seconds. The MPI library provides its own broadcast method, which is handled by the underlying system. It is supposed to be optimized, but I found the SHARCNET implementation of the MPI_Bcast method to

be unstable at best while on the *requin* cluster. At times it was very fast, and at others, it would seem to hang indefinitely. This is due to the fact that the broadcast method blocks all processors until all processors receive the broadcast. Thus if one processor is unstable or having issues, all other processors are affected [14]. I ended up manually creating a broadcast function in a binary tree structure, which was slightly slower than the MPI_Bcast at its best, but is quite consistent in that problem processors do not block other processors. This manual broadcast was used in development for other aspects of parallel BWA and for the *requin* results, but for the *orca* results, MPI_Bcast is used, since the orca cluster is more stable than *requin* and could execute an automatic broadcast without a problem. MPI_Bcast is also used in the release candidate as it is assumed the burden of ensuring a stable system is used rests on those using the product.

3.4.3 Reading the Input Sequences

After the index is read (or sometimes before, depending on the command being run), we must read in a set of short sequences that will be aligned to the index. The user will identify a FASTQ formatted file, containing any number of sequences. If there are more than 262144 sequences, BWA will perform the alignments in iterations, each iteration aligning the next 262144 sequences contained in the FASTQ file until the last iteration which will contain anywhere between 0 and 262144 sequences [19]. This is for memory management purposes. If we were to identify a FASTQ file containing 100 000 000 sequences, a processor will not be able to load every sequence into memory and will likely start paging excessively or thrashing. BWA provides the functionality needed to read gzipped (compressed) or uncompressed files via the *kseq* interface [2]. Due to the nature of compressed files, and due to the fact that BWA does not require the user to identify ahead of time whether or not the input reads file is gzipped, random file access using the *kseq* interface is impossible.

This proved to be an important issue when attempting to parallelize reading the input sequences.

BWA Much like the index reading, the sequential version of BWA only ever has one processor reading one sequence file, even when using multithreading. Reading an individual sequence is done in a simple manner. Characters are read until the beginning of a FASTQ line is recognized (with the >, @, or + characters). At this point, every character read that is not a line delimiter is considered part of either the title, sequence, or quality lines and added to the sequence's respective data structure [24]. This is done until end of file or until 262144 sequences have been read [18].

Parallelism without Communication The big advantage that parallel computing provides us in a parallel version of BWA is that we can aim to align $p \times 262144$ sequences in the time it takes the sequential version to align 262144 sequences, where p is equal to the number of processors being utilized. Obviously it would be no improvement if each of our p processors aligned the same 262144 sequences through each iteration of the program. Thus the problem of parallelizing the input of the sequences became a problem of how to efficiently divide the FASTQ file among p processors, while ensuring efficient file access is simultaneously provided. The first approach tested works as follows. For processor q, start by skipping the first $q - 1$ sequences. Then, after each sequence is read, skip through p more sequences. This approach guaranteed that the sequences were as evenly distributed as possible. However, this approach did not provide efficient file access. Each processor was essentially reading every single sequence in the file but only aligning $1/p$ of them. With p processors, it would take p times more time than it normally should to read a set of input sequences. This method was a good distributor of sequences, but a poorly performing one. A second approach tested was found to be efficient, but only works

for uncompressed files where the sequence lengths were identical for each read-pair. The *kseq* interface provided had to be circumvented, disabling the option for compressed file input. This allowed the incorporation of random file access into the solution. The solution is as follows. Each processor seeks to the end of the FASTQ file in order to obtain its size. Each processor then divides this size by p in order to obtain its block size. Each processor then seeks to byte [*(block size)*$\times$*(processor rank)*] and reads the input sequences until its block size is reached or 262144 sequences are obtained. This solution provided decent sequence distribution, with each processor only varying about 0.01% in the number of reads aligned. This solution was efficient, with each processor taking the same amount of time to read 262144 sequences as did BWA. This result may not be intuitive, as every processor is simultaneously accessing a file. When every processor simultaneously accessed the index file, some processors were blocked out, however, no blockage occurred with this strategy. This is because when reading the FASTQ file, each processor only reads 1024 characters at a time, and then *processes* the 1024 characters into sequence structures. This means that the FASTQ file access requests will be staggered among processors as they switch between reading from the file and building sequence structures out of the read data. As was mentioned earlier, this approach only works with uncompressed files as the *kseq* interface had to be circumvented. As was stated earlier, this approach is only valid for files where the sequence lengths are identical for each mate in a pair. This is due to the fact that the approach takes the total size of the file and divides by the number of processors. When each mate pair is of equal length, a processor will land on the i^{th} read for both files performing this division. However, when the mate-pairs are of differing lengths, a processor could land on different reads, which in effect ruins the program run as the *sampe* command relies on the assumption that the suffix-array index for read i in the first mate-pair file corresponds to the suffix-array index for read i in the second mate pair file.

Parallelism with Communication The same approach used to read and store the index was not considered for sequence reading, for the following reason. It can not possibly be efficient for one processor to read a number of sequences and distribute them among the remaining processors. Where the genome index is identical for each processor, the read set is not. We wish each processor to have its own set of unique reads in order to improve the efficiency of the program. Due to this fact, the controlling processor would need to load and individually communicate p times the number of reads. Thus while the controlling processor is performing file I/O, the remaining processors are idling. As a result, the following solution was developed and is used in the release candidate. The user is required to input as a parameter the number of reads in the input *FASTQ* file. Processor 0 then takes this number and quadruples it, in effect calculating the number of lines in the file. Processor 0 then skips j lines, where j = *(number of lines / number of processors)*. Each time j lines are skipped, processor 0 sends its file position to processor k, where k is the lowest ranked processor that has not yet received a file position. In effect, we are indexing the input reads on the fly. This method ensures that no matter how long each mate-pair read is, each processor is guaranteed to start on the same input read across each of the mate-pair files. This guarantees that the *sampe* command can run as it was designed. This method is fairly efficient, but not as efficient as the non-communicative method, as one processor needs to scan the entire file prior to anything else happening. However, due to the stability and flexibility of this strategy, this is the strategy used in the current parallel version. It should be noted that this approach still requires the use of uncompressed files, due to the random-access requirement of the strategy.

3.4.4 Determining the Suffix Array Intervals

After we have both the input reads and the index, the suffix array intervals can be determined. Unlike the first two functions which required much re-engineering in order to be improved by parallel computing, the speedup gained in this function is gained as a result of the parallelization done in the first two functions.

BWA Multithreading was implemented in the sequential version of BWA by Heng Li. It can be used by adding the *-t* parameter to the command line call of the BWA *aln* function followed by an integer specifying the number of threads to fork. The execution time of the program is reduced by some factor of t. When adapting BWA to include the improvements made to multithreading as specified earlier, multithreading for this function is quite efficient.

Parallelism without Communication Since determining the suffix array intervals includes reading the index and input sequences, some communication is involved even when approaching from a non-communicative standpoint. This is because the best solutions to index and sequence input were communicative approaches. When determining the suffix array intervals, each processor receives its sequence-file offset and index from a master processor via a broadcast. It should be noted that during this phase, the master processor also writes down each processor's individual file offsets into an index file. Later, during the *'sampe'* command run, the master processor can just read each processor's individual offset from the index file without having to scan the input sequence files, improving the efficiency of the *'sampe'* command. After this, each processor essentially executes the program sequentially. Each processor is assigned its own output file. The output file is a prefix (defined as a command-line parameter), followed by "-##.sai", where "##" is the processor rank. Each processor determines suffix array intervals for n reads, where n is

equal to the number of sequences specified by the user divided by the number of processors in use. Since each processor is given a starting position within the read file, every read will be processed and the reads will be evenly distributed. Each processor will dump its output into its own output file and the program will terminate upon completion.

Parallelism with Communication A communicative approach was considered to force the parallel version of BWA to behave more similarly to the sequential version. The communicative approach involves each processor writing to the same output file, so a run of the *aln* command will result in one suffix array interval file, no matter how many processors were specified. This is facilitated as follows. After each block of sequences (recall that only 262144 sequences are processed at a time for memory management) is processed, processor n communicates the number of bytes it will be writing to processor $n+1$. Processor $n+1$ sums this with the number of bytes it will write and sends the calculated sum to processor $n+2$, and so on. Once each processor has received its file offset, it outputs this byte offset to an offset file for use during subsequent commands run on the same data-set. It then outputs its suffix array interval information to the output file and continues on to the next block of reads. One complication that arose with this method is that the MPI interface needs to have a defined MPI_Datatype for all data written to file or communicated among processors. Since the sequential version of BWA has a predefined data structure for intervals, an MPI_Datatype needed to be created in order to facilitate outputting interval information in parallel to a single output file. This is done by specifying relative byte offsets of each element in the data structure and grouping the offsets into an MPI_Datatype. Due to the fact that the communicative approach, while maintaining a cleaner directory, was slower than the non-communicative approach, the non-communicative approach was used for test runs and benchmarking.

3.4.5 Determining the Alignments from the Suffix Array Intervals

Once our suffix array intervals have been determined, we must run BWA again, using the *'sampe'* or *'samse'* commands. This takes the suffix array intervals for each mate in a paired-read set and determines the best alignment. Both commands start by performing a single-end alignment. If we are running *'samse'*, we output our results and terminate. Otherwise, BWA then calculates statistical values over a large set of reads to determine the mean, standard deviation, and variance of the insert sizes between the single-end mappings of each mate-pair. It then uses the context provided by the statistical information generated and the individual mate alignments to determine if the final alignment has been properly paired. If a single-end alignment cannot be found for one of the mates then BWA uses the Smith-Waterman algorithm [41] to align it near its already-aligned mate-pair.

BWA Unlike for its *'aln'* command, BWA does not implement multithreading for the *'sampe'* command. Unofficial reasoning for this has been given by BWA author Heng Li [25]. One reason behind a non-implementation of multithreading is the *'samse'* (single-end mapping) is faster than the *'aln'* command, and thus is not as much of a bottleneck. Li also states that *'sampe'* is faster than *'aln'* when the read length is greater than fifty. Remember that the shorter our reads are, the larger our suffix array intervals will be, which results in more potential mappings having to be checked during the *'sampe'* run. Unfortunately, Illumina/Solexa reads can be as short as $36bp$ in length, and the *'sampe'* command takes longer to run than the *'aln'* command, making it in fact more of a bottleneck. The main reason multithreading is not possible, however, is that the *'sampe'* command hashes the mappings for all suffix array intervals encountered, so that when a repeat interval is encountered, the mapping can just be grabbed from a hash table instead of needing to be recalculated. It is interesting to see that as the *'sampe'* command runs, each group of

sequences is mapped faster than the previous. In a multithreaded environment, this hash table would be considered a global variable, and unfortunately would be inefficient. As soon as one thread starts reading or writing to the hash table, other threads are locked out, and must idle until the controlling thread is finished its task with the hash table. This effectively renders multithreading useless for the *'sampe'* command. Another option would be to remove the hash table altogether, but this would only make the *'sampe'* command more efficient with a large number of threads, and part of the attractiveness of BWA comes from the fact that it can be run on low-performance computers. Removing the hash table and replacing it with multithreading would only provide greater efficiency for those with expensive hardware. Li has stated[25] that a solution to this problem would be to implement a non-blocking hash table. While these are possible, this is a difficult undertaking. Another difference between the general structure of the *'sampe'* and *'aln'* commands is that of index storage. During the *'aln'* phase, the index is stored once for the duration of the phase. The *'sampe'* phase is more RAM-intensive, so the index can only be stored for certain portions of the phase and the memory used by the index must be freed before moving further into the phase. This becomes an issue when there are more than 262144 reads in a data-set, as each time we repeat the phase, we must reload the index into memory.

Parallelism without Communication Due to the lack of multithreading for *'samse'* / *'sampe'* in BWA, this is where parallelism can really start to outperform multithreading. In an MPI program, each processor has its own set of global variables, meaning each processor has its own hash table. Thus we do not face the challenge encountered by multithreading. However, a slight disadvantage caused by each processor getting its own hash table is that each processor does not have the ability to read hashed reads from the hash tables belonging to other processors. This makes the cumulative clock time taken by an MPI *'sampe'* run much larger than the clock time taken by a sequential, non-mulithreaded *'sampe'* run, as

the sequential run has access to the hashes for every single read. Another disadvantage is described above in that we must reload the index each time we process 262144 reads. This is more of a performance hit for the MPI version in comparison to the sequential version, as index loading is one of the more difficult undertakings in designing an MPI version of BWA. Since in this stage, parallel BWA creates one SAM file for every processor, the file header is only printed on the SAM file created by the processor ranked zero. These files can easily be concatenated after an alignment run by executing *cat Prefix*.sam > Master.sam*, where *Prefix* is the specified SAM file prefix. This has been found to be the most efficient way of concatenating the resulting SAM output.

Parallelism with Communication Where communication could be used to make the *'aln'* command as similar as possible across parallel/sequential platforms, it could not be used to do the same for the *'samse'/'sampe'* commands. The communicative version of the *'aln'* command was able to facilitate all processors writing to the same .sai file via communication of file offsets. This is not possible for the *'samse'/'sampe'* commands, as there is no way to tell ahead of time how much data will be output for a given read. Thus there are no communicative approaches to this command that would be able to improve, or provide any difference at all in comparison to the non-communicative approach.

3.4.6 Using Multithreading in Parallel BWA

On some parallel clusters, it is not only possible to specify how many processors you wish to run an MPI program with, but it is also possible to specify how much RAM each processor requires, and how many nodes these processors are spread across. For instance, if a system, called *SYS*, has one hundred different eight-core nodes, it is possible to specify that you want to run parallel BWA on one hundred cores with each core landing on a different node. While this is bad for parallel communication, this is extremely good in that now we are

able to utilize multithreading on top of the already efficient parallelism. Since each of our one hundred cores is on a different eight-core node, each of our one hundred cores can now execute the *'aln'* command with eight threads. The benefit of this is that we can combine the parallelism which is able to spread across multiple nodes (whereas multithreading stays on one node) with the RAM-efficient use of multithreading. On *SYS*, it might be impossible to run BWA in parallel using all 800 cores since it would require each node to have 24GB of RAM. With the multithreading combination each node only requires at most 3GB of RAM since we can use multithreading on the rest of the cores. It was thought that further benefit could be given by multithreading the single-end and paired-end alignment stages as well. This was originally considered a bad idea due to the requirement of a hash table that would be repeatedly blocked by multithreading. However, in large systems with many nodes, it is hypothesized that more efficiency could be gained by removing the hash table altogether, since each node would be given so few reads with them being spread out in a parallel execution environment. With each node aligning $1/p$ the amount of reads it normally does, the importance of the hash table drops off and multithreading can replace this. Removing the hash table from the alignment stage and replacing it with multithreading is suggested as a future work that will introduce added improvements to an already efficient parallel system. Since multithreading in the alignment stage is currently not implemented, the ratio of parallelism to multithreading must be carefully considered. Optimal use of RAM and processing time would dictate putting parallelism on one core per node and letting the rest of the cores multithread, but this will slow down the alignment stage considerably, since the alignment stage must be executed with the same number of parallel processors as the suffix array interval stage and does not allow multithreading.

4 Results

Test runs were conducted on two SHARCNET clusters, namely *requin* and *orca*. The *requin* cluster has 768 computation nodes running at 2.6GHz, each with 8GB memory and 2 cores. Since parallel BWA was developed on the *requin* cluster, it made sense to provide test data for said cluster. Since *requin* lacks many cores per node, tests were also run on *orca* to better compare parallel BWA to multithreaded BWA. The *orca* cluster has 320 computation nodes running at 2.2GHz, each with 32GB memory and 24 cores. Tests were run across datasets of 5, 25, 50, and 100 million 36bp Illumina/Solexa reads allowing two errors. An additional test was run on *orca* across a dataset of approximately 350 million 50bp SOLiD reads allowing three errors. Illumina reads were aligned to the human genome, version 18 (hg18), while SOLiD reads were aligned to the mouse genome, version 9 (mm9). All tests were conducted with paired-end reads, meaning two instances of *aln* had to be run, one for each mate in the pair. Following each *aln* run was a *sampe* run, which generates the alignments from the suffix array intervals, and takes the context paired-reads provide into account when generating said alignments. Two tables are given per test run. The first table denotes the wall-time taken to perform the test, and the second table denotes the total processor time taken, which should be approximately equal to the wall-time multiplied by the number of processors running the test. Results marked with an asterisk (*) denote counterintuitive results, such as a test run with more processors executing with less processor time than the same test run with less processors, as this shows an increase in efficiency as more processors are added. Such results can be attributed to variance in the stability and efficiency provided by a busy cluster.

4.1 Requin

The *requin* results compare BWA with 1 and 2 threads to parallel BWA with 50 and 100 processors. Due to the long wait times and busyness of the *requin* cluster, it was not possible to test with more than 100 processors. Since the *requin* nodes only have two cores, only two threads were used, and thus not much comparison between multithreaded BWA and parallel BWA can be drawn from these results. It will be seen in each of the four wall-time tables (5, 25, 50, and 100 million reads) that using 100 processors to single-threaded BWA yields speed up of 11, 23, 26, and 31, respectively. Due to the difficulty in securing processor time on *requin*, running BWA on the SOLiD data was not possible.

Illumina/Solexa - 5 Million Reads Alignment of 5 million reads showed a speedup of 11 when using 100 processors in comparison to single-threaded BWA. This shows poor scaling, as an ideal parallelization would show a speedup of 100. Poor scaling in this case can be attributed to the small read set, as more time is spent broadcasting the index and distributing the reads than is spent actually performing the alignment in parallel.

Table 2: Illumina/Solexa - 5 Million Read Wall-time Comparison [requin]

	1 Thread	**2 Threads**	**50 Processors**	**100 Processors**
Aln_1	21m, 34s	11m, 43s	2m, 13s	2m, 1s
Aln_2	20m, 53s	11m, 53s	2m, 10s	1m, 54s
Sampe	41m, 53s	41m, 53s	5m, 37s	3m, 46s
Totals	**1h, 24m**	**1h, 5m**	**10m**	**7m, 41s**

Table 3: Illumina/Solexa - 5 Million Read Processor Time Comparison [requin]

	1 Thread	**2 Threads**	**50 Processors**	**100 Processors**
Aln_1	21m, 34s	22m, 30s	1h, 50m	3h, 22m
Aln_2	20m, 53s	22m, 25s	1h, 48m	3h, 10m
Sampe	41m, 53s	41m, 53s	4h, 40m	6h, 17m
Totals	**1h, 24m**	**1h, 27m**	**8h, 19m**	**12h, 51m**

Illumina/Solexa - 25 Million Reads Alignment of 25 million reads showed a speedup of 23 when using 100 processors in comparison to single-threaded BWA. Multiplying the size of the input reads file by five seems to have improved the scale factor, as the speedup is almost twice as high as it was for the 5 million read set. This can be attributed to spending more time on parallel alignment relative to the sequential index broadcast and read distribution.

Table 4: Illumina/Solexa - 25 Million Read Wall-time Comparison [requin]

	1 Thread	**2 Threads**	**50 Processors**	**100 Processors**
Aln_1	1h, 46m	56m, 57s	5m, 40s	4m, 25s
Aln_2	1h, 50m	56m, 41s	5m, 40s	3m, 41s
Sampe	2h, 54m	2h, 54m	16m, 2s	9m, 3s
Totals	**6h, 31m**	**4h, 48m**	**27m, 22s**	**17m, 9s**

Table 5: Illumina/Solexa - 25 Million Read Processor Time Comparison [requin]

	1 Thread	**2 Threads**	**50 Processors**	**100 Processors**
Aln_1	1h, 46m	1h, 52m	4h, 43m	6h, 40m
Aln_2	1h, 50m	1h, 51m	4h, 43m	6h, 9m
Sampe	2h, 54m	2h, 54m	13h, 21m	15h, 4m
Totals	**6h, 31m**	**6h, 37m**	**22h, 48m**	**27h, 54m**

Illumina/Solexa - 50 Million Reads Alignment of 50 million reads showed a speedup of 26 when using 100 processors in comparison to single-threaded BWA. As was seen with the 25 million read set, increasing the size of our input read set improved the speedup. It is interesting to note that running BWA with two threads actually took less processor time than running BWA with one thread on this dataset. This is not an intuitive result and can be attributed to a less-than-optimal run of single-threaded BWA due to erratic cluster conditions.

Table 6: Illumina/Solexa - 50 Million Read Wall-time Comparison [requin]

	1 Thread	2 Threads	50 Processors	100 Processors
Aln_1	3h, 40m	1h, 54m	9m, 46s	6m, 28s
Aln_2	3h, 54m	1h, 54m	9m, 30s	6m, 27s
Sampe	5h, 47m	5h, 47m	25m, 32s	17m, 20s
Totals	**13h, 21m**	**9h, 35m**	**44m, 48s**	**30m, 15s**

Table 7: Illumina/Solexa - 50 Million Read Processor Time Comparison [requin]

	1 Thread	2 Threads	50 Processors	100 Processors
Aln_1	3h, 40m	3h, 44m	8h, 8m	10h, 48m
Aln_2	3h, 54m	3h, 42m	7h, 55m	10h, 45m
Sampe	5h, 47m	5h, 47m	21h, 16m	28h, 53m
Totals	**13h, 21m***	**13h, 13m***	**37h, 19m**	**50h, 26m**

Illumina/Solexa - 100 Million Reads Alignment of 100 million reads showed a speedup of 31 when using 100 processors in comparison to single-threaded BWA. As was seen with the previous read sets, increasing the size of our input read set once again improved the speedup.

Table 8: Illumina/Solexa - 100 Million Read Wall-time Comparison [requin]

	1 Thread	2 Threads	50 Processors	100 Processors
Aln_1	7h, 14m	3h, 47m	18m, 9s	11m, 20s
Aln_2	7h, 19m	3h, 46m	17m, 48s	11m, 30s
Sampe	11h, 25m	11h, 25m	41m, 13s	27m, 20s
Totals	**25h, 58m**	**18h, 58m**	**1h, 17m**	**50m, 10s**

Table 9: Illumina/Solexa - 100 Million Read Processor Time Comparison [requin]

	1 Thread	2 Threads	50 Processors	100 Processors
Aln_1	7h, 14m	7h, 24m	15h, 8m	18h, 53m
Aln_2	7h, 19m	7h, 24m	14h, 50m	19h, 9m
Sampe	11h, 25m	11h, 25m	34h, 21m	45h, 33m
Totals	**25h, 58m**	**26h, 13m**	**64h, 19m**	**83h, 35m**

4.2 Orca

The *orca* results compare BWA with 1 and 24 threads to parallel BWA with 24, 48, 96, and 240 processors. Good comparisons can be drawn by looking at BWA with 24 threads and parallel BWA with 24 processors. With the large SOLiD read set, we were able to run parallel BWA with 48 processors, each spawning 5 threads in order to compare pure parallelism (with 240 processors) to a combination of multithreading and parallelism (with 48 processors $\times$ 5 threads each). To show-off the speedup potential of parallel BWA, we also conducted a test where 240 processors each spawn 12 threads, for a total of 2880 simultaneous threads of execution. In the table headers, T is equal to the number of threads and P is equal to the number of processors. In cases where there are both T and P values, this means that each processor is spawning T threads, so the total number of threads is $P \times T$. It should be noted that in runs of multithreaded BWA, the improvements made by this work were not used, as they had not yet been officially written into a new release of BWA.

Illumina/Solexa - 5 Million Reads It can be seen that parallel BWA using 24 processors outperforms multithreaded BWA using 24 processors for both of the *aln* runs. This was an unintuitive (but positive) result, as it was assumed that the index broadcast and read distribution would add sufficient overhead to make parallel BWA less efficient. Additionally, it can be seen that parallel BWA actually slows down when moving from 96 processors to 240 processors. This can be attributed to the extra overhead incurred by having to broadcast the index to, and distribute reads to that many more processors. The speedup seen from single-threaded BWA to parallel BWA with 240 processors is 20. Like the *requin* results, this non-optimal result can be attributed to a small read set.

Table 10: Illumina/Solexa - 5 Million Read Wall-time Comparison [orca]

	1T	**24T**	**24P**	**48P**	**96P**	**240P**
Aln_1	27m, 55s	1m, 55s	1m, 39s	1m, 3s	50s	1m, 18s
Aln_2	25m, 42s	1m, 52s	1m, 33s	1m, 6s	44s	1m, 9s
Sampe	35m, 36s	35m, 36s	5m, 7s	3m, 19s	2m, 25s	2m, 3s
Totals	**1h, 29m**	**39m, 23s**	**8m, 19s**	**5m, 28s**	**3m, 59s**	**4m, 30s**

Table 11: Illumina/Solexa - 5 Million Read Processor Time Comparison [orca]

	1T	**24T**	**24P**	**48P**	**96P**	**240P**
Aln_1	27m, 55s	37m, 1s	34m, 27s	43m, 3s	1h, 4m	2h, 17m
Aln_2	25m, 42s	36m, 57s	32m, 7s	45m, 21s	59m	1h, 58m
Sampe	35m, 36s	35m, 36s	1h, 46m	2h	2h, 49m	4h, 30m
Totals	**1h, 29m**	**1h, 49m**	**2h, 53m**	**3h, 28m**	**4h, 42m**	**8h, 45m**

Illumina/Solexa - 25 Million Reads It can be seen that parallel BWA using 24 processors once again outperformed multithreaded BWA with 24 threads. An unintuitive result occurred between parallel BWA with 24 and 48 processors, as doubling the number of processors more than doubled the resultant speedup. This is theoretically impossible *assuming* that both runs were made with identical cluster conditions. It can be thusly concluded that some system lag or instability was encountered during the 24 processor run. The speedup seen from single-threaded BWA to parallel BWA with 240 processors is 45. Like the *requin* results, this improvement can be attributed to the five-fold increase in input data reads.

Table 12: Illumina/Solexa - 25 Million Read Wall-time Comparison [orca]

	1T	**24T**	**24P**	**48P**	**96P**	**240P**
Aln_1	2h, 3m	9m, 3s	8m, 31s	3m, 36s	2m, 6s	1m, 34s
Aln_2	1h, 37m	9m, 5s	8m, 31s	3m, 22s	2m, 12s	1m, 38s
Sampe	2h, 40m	2h, 40m	19m, 32s	10m, 38s	6m, 18s	5m, 12s
Totals	**6h, 20m**	**2h, 58m**	**36m, 34s**	**17m, 36s**	**10m, 36s**	**8m, 24s**

Table 13: Illumina/Solexa - 25 Million Read Processor Time Comparison [orca]

	1T	24T	24P	48P	96P	240P
Aln_1	2h, 3m	3h, 7m	3h, 2m*	2h, 23m*	2h, 52m	4h, 41m
Aln_2	1h, 37m	3h, 8m	2h, 55m*	2h, 18m*	2h, 45m	5h, 17m
Sampe	2h, 40m	2h, 40m	7h, 4m	7h, 48m	8h, 12m	14h, 8m
Totals	**6h, 20m**	**8h, 55m**	**13h, 1m***	**12h, 29m***	**13h, 49m**	**24h, 12m**

Illumina/Solexa - 50 Million Reads The speedup seen with parallel BWA using 240 processors in comparison to single-threaded BWA is 56. This is the first set of reads for which parallel BWA with 24 processors was slower than multithreaded BWA with 24 threads. The reason behind this is unclear, however it could be due to cluster instability. Given time to execute more test runs, a more definitive answer could be discovered.

Table 14: Illumina/Solexa - 50 Million Read Wall-time Comparison [orca]

	1T	24T	24P	48P	96P	240P
Aln_1	3h, 14m	17m, 42s	18m, 56s	8m, 41s	4m, 57s	2m, 59s
Aln_2	3h, 14m	18m, 6s	16m, 43s	8m, 25s	4m, 49s	2m, 55s
Sampe	6h, 35m	6h, 35m	32m, 11s	19m, 59s	14m, 38s	8m, 8s
Totals	**13h, 3m**	**7h, 11m**	**1h, 8m**	**37m, 5s**	**24m, 24s**	**14m, 2s**

Table 15: Illumina/Solexa - 50 Million Read Processor Time Comparison [orca]

	1T	24T	24P	48P	96P	240P
Aln_1	3h, 14m	6h, 7m	6h, 30m	6h, 2m	6h, 32m	9h, 2m
Aln_2	3h, 14m	6h, 14m	5h, 50m	5h, 42m	6h, 20m	9h, 25m
Sampe	6h, 35m	6h, 35m	11h, 42m	14h, 36m	18h, 31m	25h, 51m
Totals	**13h, 3m**	**18h, 56m**	**24h, 2m**	**26h, 20m**	**31h, 23m**	**44h, 18m**

Illumina/Solexa - 100 Million Reads The speedup seen with parallel BWA using 240 processors in comparison to single-threaded BWA is 57. This is marginally better than the speedup seen with 50 million reads. Better results are seen with 96 processors, where the speedup is 40. This is a better ratio, as $40/96 > 57/240$.

Table 16: Illumina/Solexa - 100 Million Read Wall-time Comparison [orca]

	1T	24T	24P	48P	96P	240P
Aln_1	7h	35m, 37s	33m, 28s	17m, 25s	9m, 9s	5m, 15s
Aln_2	8h, 53m	36m, 10s	33m, 53s	16m, 46s	9m, 19s	5m, 49s
Sampe	11h, 28m	11h, 28m	53m, 9s	32m, 3s	22m, 23s	17m, 35s
Totals	**27h, 21m**	**12h, 40m**	**2h**	**1h, 6m**	**40m, 51s**	**28m, 39s**

Table 17: Illumina/Solexa - 100 Million Read Processor Time Comparison [orca]

	1T	24T	24P	48P	96P	240P
Aln_1	7h	12h, 17m	11h, 59m	12h, 2m	12h, 34m	17h, 4m
Aln_2	8h, 53m	12h, 29m	12h, 17m	11h, 35m	12h, 34m	17h, 7m
Sampe	11h, 28m	11h, 28m	20h	23h, 30m	29h, 29m	66h, 20m
Totals	**27h, 21m**	**36h, 14m**	**44h, 16m**	**47h, 7m**	**54h, 37m**	**100h, 31m**

SOLiD - 350 Million Reads We were able to combine multithreading with parallelism with SOLiD reads. This enables the internal comparison between parallel BWA and parallel BWA combined with multithreading. When comparing 48 processors each with 5 threads to 240 processors, it appears that pure parallelism is more efficient. This can be attributed to the broadcast and distribution required by parallelism in addition to the thread spawning and merging. The combination of both the multithreaded and parallel overhead leads to a slightly less efficient test run. The positive part of the multithreaded-parallel combination is the efficient use of RAM while still allowing for parallelism. If a system does not have enough RAM per node to allow every core to run its own instance of parallel BWA, multithreading can be added in to make use of otherwise unused cores. Our most aggressive test can on this dataset, where we used 240 parallel processors, each spawning 12 threads, for a total of 2880 threads of execution. This test showed a speedup of 363 in comparison to single-threaded BWA.

Table 18: SOLiD - 350 Million Read Wall-time Comparison [orca]

	1T	**24T**	**24P**	**48P**	**48P,5T**	**96P**	**240P**	**240P,12T**
Aln_1	126h,51m	9h,30m	10h,6m	4h,54m	1h,6m	2h,20m	1h,2m	13m,21s
Aln_2	115h,50m	6h,55m	8h,15m	4h,13m	59m	2h,4m	55m,8s	12m,13s
Sampe	8h,40m	8h,40m	1h,7m	33m,42s	33m,42s	24m,9s	15m,52s	15m,52s
Totals	**251h,21m**	**25h,9m**	**19h,28m**	**9h,41m**	**2h,39m**	**4h,48m**	**2h,12m**	**41m,27s**

Table 19: SOLiD - 350 Million Read Processor Time Comparison [orca]

	1T	**24T**	**24P**	**48P**	**48P,5T**	**96P**	**240P**	**240P,12T**
Aln_1	126h,51m	212h,32m	222h,50m	214h,31m	227h,43m	200h,35m	210h,25m	232h,19m
Aln_2	115h,50m	153h,32m	187h,35m	186h,1m	205h,2m	180h,30m	194h,3m	210h,11m
Sampe	8h,40m	8h,40m	24h,43m	26h,29m	26h,29m	37h,52m	60h,17m	60h,17m
Totals	**251h,21m**	**374h,44m**	**435h,8m**	**427h,1m**	**459h,14m**	**419h,2m**	**464h,45m**	**502h,47m**

5 Discussion

5.1 Improvements by Parallelized BWA

Speed As can be seen in the tables above, parallel BWA performs quite well. It was expected that multithreaded BWA would outperform parallel BWA for the suffix array index generation stage when BWA was run with the same number of threads as parallel BWA was run with processors, but this does not seem to be the case. On the *Orca* system, 24 threads was of comparable speed to 24 processors, and parallel BWA has the extra bonus of a parallel paired-end alignment, making it that much faster than multithreaded BWA. It is hypothesized that multithreaded BWA experienced significant slowdown due to 24 threads fighting over shared resources such as read distribution and Burrows-Wheeler transform access. In effect, this shared resource competition is responsible for multithreaded BWA only showing an approximate twelve-fold increase in efficiency when using 24 threads. It should be noted that multithreaded BWA without the added improvements was used as a benchmark. It is hypothesized that parallel BWA experienced slowdown in the suffix array stage due to the initial broadcast of the Burrows-Wheeler transform and the indexing of the input sequences. As with multithreaded BWA, parallel BWA showed an approximate twelve-fold increase in efficiency with the use of 24 processors in parallel. In general, it appears that for the suffix array stage, parallel BWA shows better speedup than during the paired-end alignment stage. This is due to the fact that if we are not able to pre-load the BWT index due to lack of sufficient RAM, we need to free part of the index and re-broadcast it every set of 262144 reads. This means that once processor k is finished aligning its set of reads, it must idle until all other p-1 processors have finished aligning their reads. Only at this point can the index be re-broadcast. Thus if we cannot pre-load the BWT index, each set of 262144 is only as fast as the slowest processor. This is hypothesized to be

the reason why the paired-end alignment stage shows less of an increase in speed than the suffix array interval stage. This is illustrated by comparing the *Orca* cluster results to the *Requin* cluster results. The *Orca* cluster contains enough RAM to facilitate BWT index pre-loading while the *Requin* cluster does not. It can be seen that the results from the *Orca* cluster appear to show similar speedup across the board whereas the results from the *Requin* cluster appear to show better speedup for the suffix array stage in comparison to the paired-end alignment stage. It should be noted that index pre-loading is not supported for SOLiD reads. The reasoning behind this is unclear (I am searching out an answer but have been unsuccessful to date), but this is why the similar speedup is not seen for the SOLiD reads on *Orca*.

Scalability It appears that parallel BWA adheres to both Amdahl's law and Gustafson's law in terms of scalability. Recall that Amdahl's law states that as the ratio of sequential to parallel operations in a parallel algorithm increases, the maximum possible speedup decreases [44]. This is seen between the *aln* and *sampe* stages. The *sampe* stage shows less speedup than the *aln* stage. This is due to the fact that for moderately large input datasets, the index must be rebroadcast for each set of sequences aligned during the *sampe* stage, and thus there is a larger sequential to parallel ratio in this algorithm. Another instance of Amdahl's law can be seen when executing the same size data-set on a larger set of parallel processes. When the number of parallel processes increases, more sequential time is required to broadcast the index and index the input read set. Since the size of the data-set is constant, the ratio of sequential to parallel operations increases, and leads to slower speedup. This can be seen in any of the results tables, where increasing the number of processors decreases the wall-time taken, but increases the clock-time taken. This can be seen especially in Table 10, where increasing the number of processors from 96 to 240 actually increases execution time. Essentially, Amdahl's law assumes the parallel

computation time is constant and describes the resultant speedup in terms of variable sequential computation time. This is in contrast to Gustafon's law, which assumes the sequential time is constant and describes the resultant speedup in terms of variable paralell computation time [44]. To illustrate Gustafson's law, we increase the size of our input read file to match the increase in the size of our parallel system. Parallel BWA adheres to Gustafon's law, as illustrated between two results tables of different data-set sizes. If the data-set parallel BWA is running on doubles and the number of parallel processes stays the same, the wall-time taken will less-than-double. For this reason, there is more benefit to using parallel BWA with larger data sets, which are becoming more and more prevalent given the NGS technologies of today.

RAM It should be noted that the results between BWA and parallel BWA are not directly comparable, as parallel BWA requires as much RAM per processor as threaded BWA needs as a whole. The RAM issue can be overcome, however, by combining multithreading and parallelism as discussed earlier. When combining multithreading and parallelism, the result is slightly less efficient for the *aln* stage than if we were to use pure parallelism, as shown in Table 18, where we have 48 processors, each utilizing 5 threads. This is equal to 240 threads. The results for 48 parallel processors and 5 threads per processor are slightly less efficient than the results for 240 parallel processors without any multithreading. Of course they are even slower during the *sampe* stage, when we cannot use multithreading and are left with only 48 parallel processors.

5.2 Best Use of Parallel BWA

Parallel BWA is not meant for execution on a home computer. It is meant for large-scale systems with plenty of RAM available. To take full advantage of parallel BWA, it is

recommended for use on large data sets. For example, while a decrease in wall-time from 90 minutes to 4 minutes as seen in Table 10 is not so critical, the drop from 28 hours to 28 minutes for 100 million reads as shown in Table 16 is significant given the increasing size of files we are seeing generated from NGS technologies. With the most recent NGS platforms generating hundreds of millions of sequences in a day or even an hour [15, 10], going with a utility like parallelized BWA for alignment of these huge files seems obvious given the speedup shown in Tables 16 and 18 for the large test datasets. For systems like *requin* with many nodes that do not contain many cores, it is recommended to skip multithreading and use as much parallelism as possible. This is due to the fact that systems like these usually have enough RAM to facilitate parallelism on all cores. For systems like Orca, where there are many cores per-node and not enough RAM per core, it is recommended to use a combination of multithreading and parallelism for the *aln* stages and parallelism without multithreading for the *sampe* stage.

5.3 Future Work

Compressed input files There are a number of things that can be done in the future to improve parallel BWA even further. The first item of note is a feature that exists in BWA but had to be removed in parallel BWA. This feature is supported for compressed read files. This feature was removed from parallel BWA as it requires random file access to facilitate initial read distributions across parallel processors. At this time, there is no efficient solution to this problem. An inefficient solution exists, namely the first approach discussed in Section 3.4.3. In short, this solution consists of processor k skipping the first k reads, then for each read scanned, processor k skips p more reads. This solution is inefficient because each processor is scanning and initializing every read in the input file but only aligning $1/p$ of them. Thus the time taken to scan and initialize read sets is

multiplied by p with this solution. It is proposed that a future work could prepare a more efficient solution, or work this solution into parallel BWA with an option and disclaimer if users wish to input compressed files.

Better insert size estimation A second improvement needed was discussed earlier. That is preventing massive insert size estimations or providing a more conservative method for estimating insert sizes. A first attempt yielded decent results, but more study on statistical methods and the biology behind insert sizes is needed before any confidence can be laid in a potential solution.

Multithreading for pairing A third improvement suggested is potentially the most beneficial. The addition of multithreading to the alignment stage would be of great benefit to those systems that already benefit from a combination of parallelism and multithreading. Currently, if a system utilizes p processors and t threads, it is essentially executing the *aln* stage with $p \times t$ threads of execution. However, for the *sampe/samse* stages, the system is only able to execute with p processors, and the speedup gained is cut by a factor of t. It is proposed that removal of the hash table and introduction of multithreading would be of great benefit to parallel BWA, but this is not a trivial undertaking.

Add more functions A final improvement is based on the fact that BWA does not merely consist of the *aln*, *sampe*, and *samse* functions. There are many functions available, not the least important being the *index* function. Since these other functions work best in a sequential manner it would be best if they were executed using standard BWA. Parallel BWA could be modified to pseudo-sequentially execute these functions, but it would be a waste of processing time, as one processor would execute the functions while the other $p-1$ processors would idle. An intuitive solution would be to execute these functions with

only one processor in parallel but some systems (like SHARCNET) do not allow a parallel job to be submitted with only one processor specified.

6 Conclusion

We have created parallel BWA, an efficient parallel implementation of BWA v0.5.9. It is usable on systems with an implementation of the OpenMPI interface. It allows BWA to be run on different processors simultaneously while maintaining multithreaded abilities for the *aln* command. Each parallel processor used requires the same amount of RAM as standard BWA. It is hypothesized that adding multithreading to the alignment stage of parallel BWA would provide even further speedup. Parallel BWA makes use of the OpenMPI library to efficiently broadcast our index in order to facilitate efficient sequence alignment in parallel. When combined with multithreading, parallel BWA is a tool that can be utilized on parallel systems of all shapes and sizes. Parallel BWA is a tool that can greatly decrease the wall time required to align massive input read files to large genomes and will better facilitate analysis of the massive amount of genomic sequence data generated by NGS platforms.

References

[1] SG. Akl. *Parallel Computation: Models and Methods.* Queen's University. Kingston, ON, CA, 2003.

[2] Anonymous. What does zlib-1.2.5 bring to us? `http://attractivechaos.wordpress.com/2010/05/20/what-does-zlib-1-2-5-bring-to-us/`. Used in BWA files kseq.h, khash.h, kvec.h, and ksort.h.

[3] DR. Bentley. Whole-Genome Re-Sequencing. *Curr Opin Genet Dev*, 16(6):545–552, 2006.

[4] Daniel G. Brown. A survey of seeding for sequence alignment. *Bioinformatics Algorithms: Techniques and Applications*, 2007.

[5] M. Burrows and DJ Wheeler. A Block-sorting Lossless Data compression Algorithm. *Digital Systems Research Center*, 124:1–18, 1994.

[6] Compute-Calcul Canada. Notices for requin.sharcnet.ca. `https://www.sharcnet.ca/my/systems/status/17`.

[7] Compute-Calcul Canada. Sharcnet - shared hierarchical academic research computing network. `http://sharcnet.ca/`.

[8] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 172–181, 1992.

[9] P. Cock, C. Fields, N. Goto, M. Heuer, and P. Rice. The Sanger FASTQ File Format for Sequences with Quality Scores, and the Solexa/Illumina FASTQ Variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010.

[10] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, A. Bibillo, K. Bjornson, B. Chaudhuri, F. Christians, R. Cicero, S. Clark, R. Dalal, A. Dewinter, J. Dixon, M. Foquet, A. Gaertner, P. Hardenbol, C. Heiner, K. Hester, D. Holden, G. Kearns, X. Kong, R. Kuse, Y. Lacroix, S. Lin, P. Lundquist, C. Ma, P. Marks, M. Maxham, D. Murphy, I. Park, T. Pham, M. Phillips, J. Roy, R. Sebra, G. Shen, J. Sorenson, A. Tomaney, K. Travers, M. Trulson, J. Vieceli, J. Wegener, D. Wu, A. Yang, D. Zaccarin, P. Zhao, F. Zhong, J. Korlach, and S. Turner. Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, 323:133–138, 2008.

[11] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398, 2000.

[12] RL. Graham, GM. Shipman, BW. Barrett, RH. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A High-Performance, Heterogeneous MPI. In *IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[13] M. Hahn. Sharcnet help ticket no. 12216. `https://www.sharcnet.ca/my/problems/ticket/12216`.

[14] M. Hahn. Sharcnet help ticket no. 12689. `https://www.sharcnet.ca/my/problems/ticket/12689`.

[15] TD. Harris, PR. Buzby, H. Babcock, E. Beer, J. Bowers, I. Braslavsky, M. Causey, J. Colonell, J. DiMeo, JW. Efcavitch, E. Giladi, J. Gill, J. Healy, M. Jarosz, D. Lapen, K. Moulton, SR. Quake, K. Steinmann, E. Thayer, A. Tyurina, R. Ward, H. Weiss, and Z. Xie. Single-Molecule DNA Sequencing of a Viral Genome. *Science*, 320:106–109, 2008.

[16] WK. Hon, TW. Lam, K. Sadakane, WK. Sung, and SM. Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. *Algorithmica*, 48:23–26, 2007.

[17] B. Langmead, C. Trapnell, M. Pop, and SL. Salzberg. Ultrafast and Memory-Efficient Alignment of Short DNA Sequences to the Human Genome. *Genome Biol*, 10(3):R25, 2010.

[18] H. Li. Bwa v0.5.9, file bwaseqio.c, lines 159-198.

[19] H. Li. Bwa v0.5.9, file bwtaln.c, line 193.

[20] H. Li. Bwa v0.5.9, file bwtaln.c, line 212.

[21] H. Li. Bwa v0.5.9, file bwtaln.c, line 82.

[22] H. Li. Bwa v0.5.9, file bwtaln.c, lines 116-122.

[23] H. Li. Bwa v0.5.9, file bwtio.c, lines 51-70.

[24] H. Li. Bwa v0.5.9, file kseq.h.

[25] H. Li. Why no multithreading for bwa sampe/samse? `http://seqanswers.com/forums/showthread.php?t=4109`.

[26] H. Li and R. Durbin. Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform. *Bioinformatics*, 25(15):1754–1760, 2009.

[27] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The Sequence Alignment/map (SAM) Format and SAMtools. *Bioinformatics*, 25:2078–2079, 2009.

[28] H. Li, J. Ruan, and R. Durbin. Mapping Short DNA Sequencing Reads and Calling Variants Using Mapping Quality Scores. *Genome Research*, 18:1851–1858, 2008.

[29] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: Short Oligonucleotide Alignment Program. *Bioinformatics*, 24(5):713–714, 2008.

[30] R. Li, C. Yu, Y. Li, TW. Lam, SM. Yiu, K. Kristiansen, and J. Wang. SOAP2: an Improved Ultrafast Tool for Short Read Alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[31] KJ. McKernan, HE. Peckham, G. Costa, S. McLaughlin, E. Tsung, Y. Fu, C. Clouser, C. Dunkan, J. Ichikawa, C. Lee, Z. Zhang, A. Sheridan, H. Fu, S. Ranade, E. Dimilanta, T. Sokolsky, L. Zhang, C. Hendrickson, B. Li, L. Kotler, J. Stuart, J. Malek, J. Manning, A. Antipova, D. Perez, M. Moore, K. Hayashibara, M. Lyons, R. Beaudoin, B. Coleman, M. Laptewicz, A. Sanicandro, M. Rhodes, F. De La Vega, RK. Gottimukkala, F. Hyland, M. Reese, S. Yang, V. Bafna, A. Bashir, A. MacBride, C. Aklan, J. Kidd, EE. Eichler, and AP. Blanchard. Sequence and Structural Variation in a Human Genome Uncovered by Short-Read, Massively Parallel Ligation Sequencing Using Two-Base Encoding. *Genome Research*, 19(9):1527–1541, 2009.

[32] ML. Metzker. Sequencing Technologies - The Next Generation. *Nature Rev. Genet.*, 11:31–46, 2010.

[33] G. Myers. A Sublinear Algorithm for Approximate Keyword Searching. *Algorithmica*, 12(4):345–374, 1994.

[34] G. Myers. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *J. ACM*, 46(3):395–415, 1999.

[35] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[36] G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000.

[37] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[38] E. Pettersson, J. Lundeberg, and A. Ahmadian. Generations of Sequencing Technologies. *Genomics*, 93(2):105–111, 2009.

[39] F. Sanger, S. Nicklen, and AR. Coulson. DNA Sequencing with Chain-terminating Inhibitors. *Proc. Natl. Acad. Sci. U.S.A.*, 74(12):5463–5467, 1977.

[40] P. Sellers. The Theory and Computation of Evolutionary Distance: Pattern Recognition. *J. Algor.*, 1:359–393, 1980.

[41] TF. Smith and MS. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[42] A. Tanenbaum. *Modern Operating Systems.* Pretence Hall, Upper Saddle River, New Jersey, 2 edition, 2001.

[43] E. Ukkonen. Finding Approximate Patterns in Strings. *J. of Algorithms*, 6:132–137, 1985.

[44] B. Wilkinson and M. Allen. *Parallel Programming.* Pretence Hall, Upper Saddle River, New Jersey, 2 edition, 2005.

[45] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Common. ACM*, 35(10):83–91, 1992.

[46] D. Zerbino and E. Birney. Velvet: Algorithms for De Novo Short Read Assembly Using de Bruijn Graphs. *Genome Research*, 18(5):821–829, 2008.

[47] Z. Zheng, A. Advani, O. Melefors, S. Glavas, H. Nordstrm, W. Ye, L. Engstrand, and AF. Andersson. Titration-Free Massively Parallel Pyrosequencing Using Trace Amounts of Starting Material. *Nucleic Acids Res*, 38(13):e137, 2010.

7 Appendix

7.1 List of Modifications to BWA

The below is output from subversion, a command-line code revision tool. I ran subversion between BWA 0.5.9 and a working copy of parallel BWA.

```
Index: bwt_gen/libbwtgen.a
===================================================================
Cannot display: file marked as a binary type.
svn:mime-type = application/octet-stream
Index: bwaseqio.c
===================================================================
--- bwaseqio.c (revision 1)
+++ bwaseqio.c (working copy)
@@ -1,3 +1,6 @@
+#include <mpi.h>
+#include <time.h>
+#include <stdio.h>
 #include <zlib.h>
 #include <ctype.h>
 #include "bwtaln.h"
@@ -5,7 +8,8 @@
 #include "bamlite.h"

 #include "kseq.h"
-KSEQ_INIT(gzFile, gzread)
+//here we modify kseq - from compressed to uncompressed files
+KSEQ_INIT(FILE*, fread)

 extern unsigned char nst_nt4_table[256];
 static char bam_nt16_nt4_table[] = { 4, 0, 1, 4, 2, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4 };
@@ -31,13 +35,65 @@
  return bs;
 }

-bwa_seqio_t *bwa_seq_open(const char *fn)
+bwa_seqio_t *bwa_seq_open(const char *fn, long long int numToRead, char *outIndex)
 {
-gzFile fp;
+ FILE * fp;
+ FILE * file;
+ int rank, size, i;
+ long long int numLines = 0;
+ long long int pos;
+ char c[500];
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+ MPI_Comm_size(MPI_COMM_WORLD, &size);
+
  bwa_seqio_t *bs;
  bs = (bwa_seqio_t*)calloc(1, sizeof(bwa_seqio_t));
-fp = xzopen(fn, "r");
+ fp = xopen(fn, "r");
+
  bs->ks = kseq_init(fp);
+ //processor 0 controls the sequence distribution
+ if (rank == 0) {
```

```
+ //first we see if our reads file is indexed
+ if ((file = fopen(outIndex, "r"))) {
+
+ fprintf(stderr, "Distributing input reads... ");
+ //it is!  So we simply read from the index and send the values out
+ for (i = 1; i < size; i++) {
+
+ fread(&pos, 8, 1, file);
+ MPI_Send(&pos, 1, MPI_LONG_LONG_INT, i, 0, MPI_COMM_WORLD);
+ }
+ fprintf(stderr, "done!\n");
+ } else {
+ //it is not indexed yet, so we index the file and send the values out
+ file = fopen(outIndex, "w");
+ fprintf(stderr, "Indexing reads file... ");
+ for (i = 1; i < size; i++) {
+
+ for (numLines = 0; numLines < numToRead*4; numLines++) {
+
+ fgets(c, 500, bs->ks->f->f);
+ }
+ pos = ftello64(bs->ks->f->f);
+ fwrite(&pos, 8, 1, file);
+ MPI_Send(&pos, 1, MPI_LONG_LONG_INT, i, 0, MPI_COMM_WORLD);
+ }
+
+ fseeko64(bs->ks->f->f, 0, SEEK_SET);
+ }
+
+ fclose(file);
+ //we are not processor 0, so we receive out value from processor 0
+ } else {
+
+ MPI_Recv(&pos, 1, MPI_LONG_LONG_INT, 0, 0, MPI_COMM_WORLD, NULL);
+ fseeko64(bs->ks->f->f, pos, SEEK_SET);
+ }
+
+ if (rank == 0) {
+
+ fprintf(stderr, "done!\n");
+ }
  return bs;
 }

@@ -46,7 +102,7 @@
  if (bs == 0) return;
  if (bs->is_bam) bam_close(bs->fp);
  else {
-gzclose(bs->ks->f->f);
+ fclose(bs->ks->f->f);
  kseq_destroy(bs->ks);
  }
  free(bs);
@@ -141,9 +197,12 @@
 }

 #define BARCODE_LOW_QUAL 13
-
```

```
-bwa_seq_t *bwa_read_seq(bwa_seqio_t *bs, int n_needed, int *n, int mode, int trim_qual)
+//we add 'tot_seqs' and 'numToRead' to our definition.  This is for read distribution
+bwa_seq_t *bwa_read_seq(bwa_seqio_t *bs, int n_needed, int *n, int mode, int trim_qual, int tot_seqs, long long int numToR
 {
+ /*In BWA, normally we stop at EOF.  Since we don't have that convenience, we need to pass numToRead (num in file/size)
+ ** and the number of seqs we've processed thus far.  If we've done our bit, we're done.  Return 0. */
+ if (tot_seqs >= numToRead) return 0;
  bwa_seq_t *seqs, *p;
  kseq_t *seq = bs->ks;
  int n_seqs, l, i, is_comp = mode&BWA_MODE_COMPREAD, is_64 = mode&BWA_MODE_IL13, l_bc = mode>>24;
@@ -195,7 +254,8 @@
  int t = strlen(p->name);
  if (t > 2 && p->name[t-2] == '/' && (p->name[t-1] == '1' || p->name[t-1] == '2')) p->name[t-2] = '\0';
  }
-if (n_seqs == n_needed) break;
+ //if we've read all the sequences we need to process, the get out of here!
+ if (n_seqs == n_needed || (n_seqs + tot_seqs) >= numToRead) break;
  }
  *n = n_seqs;
  if (n_seqs && trim_qual >= 1)
Index: bwtaln.c
===================================================================
--- bwtaln.c (revision 1)
+++ bwtaln.c (working copy)
@@ -1,3 +1,5 @@
+#include <mpi.h>
+#include "sys/time.h"
 #include <stdio.h>
 #include <unistd.h>
 #include <math.h>
@@ -11,11 +13,11 @@
 #include "bwtaln.h"
 #include "bwtgap.h"
 #include "utils.h"
+#include "bwt.h"

 #ifdef HAVE_PTHREAD
 #define THREAD_BLOCK_SIZE 1024
 #include <pthread.h>
-static pthread_mutex_t g_seq_lock = PTHREAD_MUTEX_INITIALIZER;
 #endif

 gap_opt_t *gap_init_opt()
@@ -79,6 +81,7 @@

 void bwa_cal_sa_reg_gap(int tid, bwt_t *const bwt[2], int n_seqs, bwa_seq_t *seqs, const gap_opt_t *opt)
 {
+ //note max_l = -1, this is 0 in original version
  int i, max_l = -1, max_len;
  gap_stack_t *stack;
  bwt_width_t *w[2], *seed_w[2];
@@ -96,21 +99,11 @@
  seed_w[1] = (bwt_width_t*)calloc(opt->seed_len+1, sizeof(bwt_width_t));
  w[0] = w[1] = 0;
  for (i = 0; i != n_seqs; ++i) {
-bwa_seq_t *p = seqs + i;
 #ifdef HAVE_PTHREAD
-if (opt->n_threads > 1) {
```

```
-pthread_mutex_lock(&g_seq_lock);
-if (p->tid < 0) { // unassigned
-int j;
-for (j = i; j < n_seqs && j < i + THREAD_BLOCK_SIZE; ++j)
-seqs[j].tid = tid;
-} else if (p->tid != tid) {
-pthread_mutex_unlock(&g_seq_lock);
-continue;
-}
-pthread_mutex_unlock(&g_seq_lock);
-}
+ if ((i % opt->n_threads) != tid) continue;
 #endif
+ bwa_seq_t *p = seqs + i;
+
  p->sa = 0; p->type = BWA_TYPE_NO_MATCH; p->c1 = p->c2 = 0; p->n_aln = 0; p->aln = 0;
  seq[0] = p->seq; seq[1] = p->rseq;
  if (max_l < p->len) {
@@ -156,7 +149,7 @@
 }
 #endif

-bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa)
+bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa, long long int numToRead, char *outPref)
 {
  bwa_seqio_t *ks;
  if (mode & BWA_MODE_BAM) { // open BAM
@@ -166,12 +159,20 @@
  if (mode & BWA_MODE_BAM_READ2) which |= 2;
  if (which == 0) which = 7; // then read all reads
  ks = bwa_bam_open(fn_fa, which);
-} else ks = bwa_seq_open(fn_fa);
+ } else ks = bwa_seq_open(fn_fa, numToRead, outPref);
  return ks;
 }

-void bwa_aln_core(const char *prefix, const char *fn_fa, const gap_opt_t *opt)
+void bwa_aln_core(const char *prefix, const char *fn_fa, const gap_opt_t *opt, long long int numReads, char *outPref)
 {
+ int rank, size;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+ MPI_Comm_size(MPI_COMM_WORLD, &size);
+ time_t s,f;
+ struct timeval tv;
+ //calculate the number of sequences each processor has to read
+ long long int numToRead = (long long int)(numReads/size);
+
  int i, n_seqs, tot_seqs = 0;
  bwa_seq_t *seqs;
  bwa_seqio_t *ks;
@@ -179,7 +180,8 @@
  bwt_t *bwt[2];

  // initialization
-ks = bwa_open_reads(opt->mode, fn_fa);
+ //we pass the number of reads each processor needs + our index filename now
+ ks = bwa_open_reads(opt->mode, fn_fa, numToRead, outPref);
```

```
  { // load BWT
  char *str = (char*)calloc(strlen(prefix) + 10, 1);
@@ -190,12 +192,13 @@

  // core loop
  fwrite(opt, sizeof(gap_opt_t), 1, stdout);
-while ((seqs = bwa_read_seq(ks, 0x40000, &n_seqs, opt->mode, opt->trim_qual)) != 0) {
+ while ((seqs = bwa_read_seq(ks, 0x40000, &n_seqs, opt->mode, opt->trim_qual, tot_seqs, numToRead)) != 0) {
  tot_seqs += n_seqs;
-t = clock();
+
+ gettimeofday(&tv, NULL);
+ s = (tv.tv_sec*1000) + (tv.tv_usec/1000);
+ fprintf(stderr, "Proc %d: [bwa_aln_core] calculate SA coordinate...\n", rank);

-fprintf(stderr, "[bwa_aln_core] calculate SA coordinate... ");
-
 #ifdef HAVE_PTHREAD
  if (opt->n_threads <= 1) { // no multi-threading at all
  bwa_cal_sa_reg_gap(0, bwt, n_seqs, seqs, opt);
@@ -220,19 +223,21 @@
  bwa_cal_sa_reg_gap(0, bwt, n_seqs, seqs, opt);
 #endif

-fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
+ gettimeofday(&tv, NULL);
+ f = (tv.tv_sec*1000) + (tv.tv_usec/1000);
+ fprintf(stderr, "Proc %d: [bwa_aln_core] Done SA Co-ords in %.2f sec\n", rank, (float)(f-s)/1000);

  t = clock();
-fprintf(stderr, "[bwa_aln_core] write to the disk... ");
+ fprintf(stderr, "Proc %d: [bwa_aln_core] write to the disk...\n", rank);
  for (i = 0; i < n_seqs; ++i) {
  bwa_seq_t *p = seqs + i;
  fwrite(&p->n_aln, 4, 1, stdout);
  if (p->n_aln) fwrite(p->aln, sizeof(bwt_aln1_t), p->n_aln, stdout);
  }
-fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
+ fprintf(stderr, "Proc %d: [bwa_aln_core] wrote to disk in %.2f sec\n", rank, (float)(clock() - t) / CLOCKS_PER_SEC); t =

  bwa_free_read_seq(n_seqs, seqs);
-fprintf(stderr, "[bwa_aln_core] %d sequences have been processed.\n", tot_seqs);
+ fprintf(stderr, "Proc %d: [bwa_aln_core] %d sequences have been processed.\n", rank, tot_seqs);
  }

  // destroy
@@ -244,6 +249,15 @@
 {
  int c, opte = -1;
  gap_opt_t *opt;
+ int fm = 0;
+ char filename[50];
+ char filePref[50];
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+ time_t s,f;
+ struct timeval tv;
+ gettimeofday(&tv, NULL);
```

```
+ s = (tv.tv_sec*1000) + (tv.tv_usec/1000);

  opt = gap_init_opt();
  while ((c = getopt(argc, argv, "n:o:e:i:d:l:k:cLR:m:t:NM:O:E:q:f:b012IB:")) >= 0) {
@@ -268,7 +282,8 @@
  case 'q': opt->trim_qual = atoi(optarg); break;
  case 'c': opt->mode &= ~BWA_MODE_COMPREAD; break;
  case 'N': opt->mode |= BWA_MODE_NONSTOP; opt->max_top2 = 0x7fffffff; break;
-case 'f': xreopen(optarg, "wb", stdout); break;
+ /*Create processor rank-specific output files and our input read index filename.*/
+ case 'f': fm = 1; sprintf(filename, "%s-%d.sai", optarg, rank); sprintf(filePref, "%s.ind", optarg); xreopen(filename, "w
  case 'b': opt->mode |= BWA_MODE_BAM; break;
  case '0': opt->mode |= BWA_MODE_BAM_SE; break;
  case '1': opt->mode |= BWA_MODE_BAM_READ1; break;
@@ -283,9 +298,9 @@
  opt->mode &= ~BWA_MODE_GAPE;
  }

-if (optind + 2 > argc) {
+ if (optind + 3 > argc) {
  fprintf(stderr, "\n");
-fprintf(stderr, "Usage:   bwa aln [options] <prefix> <in.fq>\n\n");
+ fprintf(stderr, "Usage:   bwa aln -f <output_file_prefix> [options] <prefix> <in.fq> <num_reads_in_in.fq>\n\n");
  fprintf(stderr, "Options: -n NUM    max #diff (int) or missing prob under %.2f err rate (float) [%.2f]\n",
  BWA_AVG_ERR, opt->fnr);
  fprintf(stderr, "         -o INT    maximum number or fraction of gap opens [%d]\n", opt->max_gapo);
@@ -301,7 +316,7 @@
  fprintf(stderr, "         -E INT    gap extension penalty [%d]\n", opt->s_gape);
  fprintf(stderr, "         -R INT    stop searching when there are >INT equally best hits [%d]\n", opt->max_top2);
  fprintf(stderr, "         -q INT    quality threshold for read trimming down to %dbp [%d]\n", BWA_MIN_RDLEN, opt->trim_qu
-        fprintf(stderr, "         -f FILE   file to write output to instead of stdout\n");
+        fprintf(stderr, "            -f FILE   file prefix for each processor's .sai output\n");
  fprintf(stderr, "         -B INT    length of barcode\n");
  fprintf(stderr, "         -c        input sequences are in the color space\n");
  fprintf(stderr, "         -L        log-scaled gap penalty for long deletions\n");
@@ -314,6 +329,13 @@
  fprintf(stderr, "\n");
  return 1;
  }
+
+ if (fm == 0) {
+
+ fprintf(stderr, "You must specify -f.\n");
+ return 1;
+ }
+
  if (opt->fnr > 0.0) {
  int i, k;
  for (i = 17, k = 0; i <= 250; ++i) {
@@ -322,8 +344,18 @@
  k = 1;
  }
  }
-bwa_aln_core(argv[optind], argv[optind+1], opt);
+ /*here we pass two extra parameters to the main function
+ **The first is the number of reads in the file, the second is
+ **our Input read index filename */
+ bwa_aln_core(argv[optind], argv[optind+1], opt, atoll(argv[optind+2]), filePref);
```

```
+ //this is my own timing function that uses wall time instead of clock time.
+ //it is more accurate, especially when multithreading is used
+ gettimeofday(&tv, NULL);
+ f = (tv.tv_sec*1000)+(tv.tv_usec/1000);
+ fprintf(stderr, "Proc %d: Total time taken: %.2f sec\n", rank, (float)(f-s)/1000);
  free(opt);
+ //we must call MPI_Finalize at the end
+ MPI_Finalize();
  return 0;
 }

Index: bwase.c
===================================================================
--- bwase.c (revision 1)
+++ bwase.c (working copy)
@@ -1,3 +1,5 @@
+#include <mpi.h>
+#include "sys/time.h"
 #include <unistd.h>
 #include <string.h>
 #include <stdio.h>
@@ -586,9 +588,9 @@
  return 0;
 }

-void bwa_sai2sam_se_core(const char *prefix, const char *fn_sa, const char *fn_fa, int n_occ)
+void bwa_sai2sam_se_core(const char *prefix, const char *fn_sa, const char *fn_fa, int n_occ, long long int numReads)
 {
-extern bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa);
+ extern bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa, long long int numToRead, char *filePref);
  int i, n_seqs, tot_seqs = 0, m_aln;
  bwt_aln1_t *aln = 0;
  bwa_seq_t *seqs;
@@ -597,12 +599,21 @@
  bntseq_t *bns, *ntbns = 0;
  FILE *fp_sa;
  gap_opt_t opt;
+ int rank, size;
+ char filePref[50];
+ char filename[50];
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+ MPI_Comm_size(MPI_COMM_WORLD, &size);
+ sprintf(filePref, "%s.ind", fn_sa);
+ sprintf(filename, "%s-%d.sai", fn_sa, rank);
+
+ long long int numToRead = (long long int)(numReads/size);

  // initialization
  bwase_initialize();
  bns = bns_restore(prefix);
  srand48(bns->seed);
-fp_sa = xopen(fn_sa, "r");
+ fp_sa = xopen(filename, "r");

  m_aln = 0;
  fread(&opt, sizeof(gap_opt_t), 1, fp_sa);
@@ -611,9 +622,9 @@
  bwa_print_sam_SQ(bns);
```

```
  bwa_print_sam_PG();
  // set ks
-ks = bwa_open_reads(opt.mode, fn_fa);
+ ks = bwa_open_reads(opt.mode, fn_fa, numToRead, filePref);
  // core loop
-while ((seqs = bwa_read_seq(ks, 0x40000, &n_seqs, opt.mode, opt.trim_qual)) != 0) {
+ while ((seqs = bwa_read_seq(ks, 0x40000, &n_seqs, opt.mode, opt.trim_qual, tot_seqs, numToRead)) != 0) {
  tot_seqs += n_seqs;
  t = clock();

@@ -630,15 +641,15 @@
  bwa_aln2seq_core(n_aln, aln, p, 1, n_occ);
  }

-fprintf(stderr, "[bwa_aln_core] convert to sequence coordinate... ");
+ fprintf(stderr, "[bwa_aln_core] convert to sequence coordinate...\n");
  bwa_cal_pac_pos(prefix, n_seqs, seqs, opt.max_diff, opt.fnr); // forward bwt will be destroyed here
  fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();

-fprintf(stderr, "[bwa_aln_core] refine gapped alignments... ");
+ fprintf(stderr, "[bwa_aln_core] refine gapped alignments...\n");
  bwa_refine_gapped(bns, n_seqs, seqs, 0, ntbns);
  fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();

-fprintf(stderr, "[bwa_aln_core] print alignments... ");
+ fprintf(stderr, "[bwa_aln_core] print alignments...\n");
  for (i = 0; i < n_seqs; ++i)
  bwa_print_sam1(bns, seqs + i, 0, opt.mode, opt.max_top2);
  fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
@@ -657,7 +668,11 @@

 int bwa_sai2sam_se(int argc, char *argv[])
 {
+ char filename[50];
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int c, n_occ = 3;
+ int fm = 0;
  while ((c = getopt(argc, argv, "hn:f:r:")) >= 0) {
  switch (c) {
  case 'h': break;
@@ -668,16 +683,23 @@
  }
  break;
  case 'n': n_occ = atoi(optarg); break;
-case 'f': xreopen(optarg, "w", stdout); break;
+ case 'f': fm = 1; sprintf(filename, "%s%d.sam", optarg, rank); xreopen(filename, "w", stdout); break;
  default: return 1;
  }
  }

-if (optind + 3 > argc) {
-fprintf(stderr, "Usage: bwa samse [-n max_occ] [-f out.sam] [-r RG_line] <prefix> <in.sai> <in.fq>\n");
+ if (optind + 4 > argc) {
+ fprintf(stderr, "Usage: bwa samse [-n max_occ] -f prefix.sam [-r RG_line] <prefix> <in.sai> <in.fq> <num_reads_in_in.fq>\
  return 1;
  }
-bwa_sai2sam_se_core(argv[optind], argv[optind+1], argv[optind+2], n_occ);
```

```
+
+ if (!fm) {
+
+ fprintf(stderr, "You must specify -f.\n");
+ return 1;
+ }
+ bwa_sai2sam_se_core(argv[optind], argv[optind+1], argv[optind+2], n_occ, atoll(argv[optind + 3]));
  free(bwa_rg_line); free(bwa_rg_id);
+ MPI_Finalize();
  return 0;
 }
Index: bwtio.c
===================================================================
--- bwtio.c (revision 1)
+++ bwtio.c (working copy)
@@ -1,3 +1,4 @@
+#include <mpi.h>
 #include <string.h>
 #include <stdio.h>
 #include <stdlib.h>
@@ -3,4 +4,5 @@
 #include "bwt.h"
 #include "utils.h"
+#include <math.h>

 void bwt_dump_bwt(const char *fn, const bwt_t *bwt)
@@ -31,6 +33,8 @@
  char skipped[256];
  FILE *fp;
  bwtint_t primary;
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  fp = xopen(fn, "rb");
  fread(&primary, sizeof(bwtint_t), 1, fp);
@@ -43,9 +47,24 @@
  bwt->n_sa = (bwt->seq_len + bwt->sa_intv) / bwt->sa_intv;
  bwt->sa = (bwtint_t*)calloc(bwt->n_sa, sizeof(bwtint_t));
  bwt->sa[0] = -1;
-
-fread(bwt->sa + 1, sizeof(bwtint_t), bwt->n_sa - 1, fp);
-fclose(fp);
+ /*all of our processors can do individual reads up until this point because
+ **the reads have been small.  This next step is reading the entire Suffix Array.
+ **Processor 0 will read it and broadcast it */
+ if (rank != 0) {
+
+ fclose(fp);
+ } else {
+
+ fprintf(stderr, "Broadcasting SA... ");
+ fread(bwt->sa + 1, sizeof(bwtint_t), bwt->n_sa - 1, fp);
+ fclose(fp);
+ }
+ MPI_Bcast(bwt->sa + 1, bwt->n_sa - 1, MPI_INT, 0, MPI_COMM_WORLD);
+
+ if (rank == 0) {
+
```

```
+ fprintf(stderr, "done!\n");
+ }
 }

 bwt_t *bwt_restore_bwt(const char *fn)
@@ -53,19 +72,43 @@
  bwt_t *bwt;
  FILE *fp;

-bwt = (bwt_t*)calloc(1, sizeof(bwt_t));
-fp = xopen(fn, "rb");
-fseek(fp, 0, SEEK_END);
-bwt->bwt_size = (ftell(fp) - sizeof(bwtint_t) * 5) >> 2;
-bwt->bwt = (uint32_t*)calloc(bwt->bwt_size, 4);
-fseek(fp, 0, SEEK_SET);
-fread(&bwt->primary, sizeof(bwtint_t), 1, fp);
-fread(bwt->L2+1, sizeof(bwtint_t), 4, fp);
-fread(bwt->bwt, 4, bwt->bwt_size, fp);
-bwt->seq_len = bwt->L2[4];
-fclose(fp);
-bwt_gen_cnt_table(bwt);
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+
+
+ bwt = (bwt_t*)calloc(1, sizeof(bwt_t));
+
+ //processor 0 will read in the BWT information and broadcast it
+ if (rank == 0) {
+
+ fprintf(stderr, "Broadcasting BWT (this may take a while)... ");
+ fp = xopen(fn, "rb");
+ fseek(fp, 0, SEEK_END);
+ bwt->bwt_size = (ftell(fp) - sizeof(bwtint_t) * 5) >> 2;
+ bwt->bwt = (uint32_t*)calloc(bwt->bwt_size, 4);
+ fseek(fp, 0, SEEK_SET);
+ fread(&bwt->primary, sizeof(bwtint_t), 1, fp);
+ fread(bwt->L2+1, sizeof(bwtint_t), 4, fp);
+ fread(bwt->bwt, 4, bwt->bwt_size, fp);
+ bwt->seq_len = bwt->L2[4];
+ fclose(fp);
+ }
+
+ MPI_Bcast(&bwt->bwt_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
+ if (rank != 0) bwt->bwt = (uint32_t*)calloc(bwt->bwt_size, 4);
+ MPI_Bcast(bwt->bwt, bwt->bwt_size, MPI_INT, 0, MPI_COMM_WORLD);
+
+ MPI_Bcast(&bwt->primary, 1, MPI_INT, 0, MPI_COMM_WORLD);
+ MPI_Bcast(bwt->L2+1, 4, MPI_INT, 0, MPI_COMM_WORLD);
+ if (rank != 0) {
+
+ bwt->seq_len = bwt->L2[4];
+ } else {

+ fprintf(stderr, "done!\n");
+ }
+
+ bwt_gen_cnt_table(bwt);
```

```
  return bwt;
 }

Index: bwape.c
===================================================================
--- bwape.c (revision 1)
+++ bwape.c (working copy)
@@ -1,3 +1,5 @@
+#include <mpi.h>
+#include "sys/time.h"
 #include <unistd.h>
 #include <math.h>
 #include <stdlib.h>
@@ -59,6 +61,7 @@
   po->type = BWA_PET_STD;
   po->is_sw = 1;
   po->ap_prior = 1e-5;
+ po->set_max = 0;
   return po;
 }

@@ -88,8 +91,10 @@
 // for normal distribution, this is about 3std
 #define OUTLIER_BOUND 2.0

-static int infer_isize(int n_seqs, bwa_seq_t *seqs[2], isize_info_t *ii, double ap_prior, int64_t L)
+static int infer_isize(int n_seqs, bwa_seq_t *seqs[2], isize_info_t *ii, double ap_prior, int64_t L, const pe_opt_t *opt)
 {
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   uint64_t x, *isizes, n_ap = 0;
   int n, i, tot, p25, p75, p50, max_len = 1, tmp;
   double skewness = 0.0, kurtosis = 0.0, y;
@@ -108,7 +113,7 @@
   if (p[1]->len > max_len) max_len = p[1]->len;
   }
   if (tot < 20) {
-fprintf(stderr, "[infer_isize] fail to infer insert size: too few good pairs\n");
+ fprintf(stderr, "Proc %d: [infer_isize] fail to infer insert size: too few good pairs\n", rank);
   free(isizes);
   return -1;
   }
@@ -142,19 +147,27 @@
   ii->ap_prior = .01 * (n_ap + .01) / tot;
   if (ii->ap_prior < ap_prior) ii->ap_prior = ap_prior;
   free(isizes);
-fprintf(stderr, "[infer_isize] (25, 50, 75) percentile: (%d, %d, %d)\n", p25, p50, p75);
+ fprintf(stderr, "Proc %d: [infer_isize] (25, 50, 75) percentile: (%d, %d, %d)\n",rank, p25, p50, p75);
   if (isnan(ii->std) || p75 > 100000) {
   ii->low = ii->high = ii->high_bayesian = 0; ii->avg = ii->std = -1.0;
-fprintf(stderr, "[infer_isize] fail to infer insert size: weird pairing\n");
+ fprintf(stderr, "Proc %d: [infer_isize] fail to infer insert size: weird pairing\n", rank);
   return -1;
   }
   for (y = 1.0; y < 10.0; y += 0.01)
   if (.5 * erfc(y / M_SQRT2) < ap_prior / L * (y * ii->std + ii->avg)) break;
   ii->high_bayesian = (bwtint_t)(y * ii->std + ii->avg + .499);
-fprintf(stderr, "[infer_isize] low and high boundaries: %d and %d for estimating avg and std\n", ii->low, ii->high);
```

```
-fprintf(stderr, "[infer_isize] inferred external isize from %d pairs: %.3lf +/- %.3lf\n", n, ii->avg, ii->std);
-fprintf(stderr, "[infer_isize] skewness: %.3lf; kurtosis: %.3lf; ap_prior: %.2e\n", skewness, kurtosis, ii->ap_prior);
-fprintf(stderr, "[infer_isize] inferred maximum insert size: %d (%.2lf sigma)\n", ii->high_bayesian, y);
+ fprintf(stderr, "Proc %d: [infer_isize] low and high boundaries: %d and %d for estimating avg and std\n", rank, ii->low,
+ fprintf(stderr, "Proc %d: [infer_isize] inferred external isize from %d pairs: %.3lf +/- %.3lf\n", rank, n, ii->avg, ii->
+ fprintf(stderr, "Proc %d: [infer_isize] skewness: %.3lf; kurtosis: %.3lf; ap_prior: %.2e\n", rank, skewness, kurtosis, ii
+ fprintf(stderr, "Proc %d: [infer_isize] inferred maximum insert size: %d (%.2lf sigma)\n", rank, ii->high_bayesian, y);
+ //if our high value is greater than our specified -a, discard isizes
+ if (opt->set_max && ii->high > opt->max_isize) {
+
+ ii->low = ii->high = ii->high_bayesian =0; ii->avg = ii->std = -1.0;
+ fprintf(stderr, "Proc %d: [infer_isize] inferred maximum insert size greater than -a, discarding isizes\n", rank);
+ }
+
+
  return 0;
 }

@@ -323,7 +336,7 @@
  }

  // infer isize
-infer_isize(n_seqs, seqs, ii, opt->ap_prior, bwt[0]->seq_len);
+ infer_isize(n_seqs, seqs, ii, opt->ap_prior, bwt[0]->seq_len, opt);
  if (ii->avg < 0.0 && last_ii->avg > 0.0) *ii = *last_ii;
  if (opt->force_isize) {
  fprintf(stderr, "[%s] discard insert size estimate as user's request.\n", __func__);
@@ -642,21 +655,28 @@
  return pacseq;
 }

-void bwa_sai2sam_pe_core(const char *prefix, char *const fn_sa[2], char *const fn_fa[2], pe_opt_t *popt)
+void bwa_sai2sam_pe_core(const char *prefix, char *const fn_sa[2], char *const fn_fa[2], pe_opt_t *popt, long long int num
 {
-extern bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa);
+ extern bwa_seqio_t *bwa_open_reads(int mode, const char *fn_fa, long long int numToRead, char *filePref);
  int i, j, n_seqs, tot_seqs = 0;
  bwa_seq_t *seqs[2];
  bwa_seqio_t *ks[2];
  clock_t t;
  bntseq_t *bns, *ntbns = 0;
  FILE *fp_sa[2];
+ char filename[2][50];
+ char filePref[50];
  gap_opt_t opt, opt0;
  khint_t iter;
  isize_info_t last_ii; // this is for the last batch of reads
  char str[1024];
  bwt_t *bwt[2];
  uint8_t *pac;
+ int size, rank;
+ MPI_Comm_size(MPI_COMM_WORLD, &size);
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
+ //calculate number of reads each processor aligns
+ long long int numToRead  = (long long int)(numReads/size);

  // initialization
  bwase_initialize(); // initialize g_log_n[] in bwase.c
```

```
@@ -664,16 +684,23 @@
  for (i = 1; i != 256; ++i) g_log_n[i] = (int)(4.343 * log(i) + 0.5);
  bns = bns_restore(prefix);
  srand48(bns->seed);
-fp_sa[0] = xopen(fn_sa[0], "r");
-fp_sa[1] = xopen(fn_sa[1], "r");
+ //create our processor rank-specified filenames
+ sprintf(filename[0], "%s-%d.sai", fn_sa[0], rank);
+ sprintf(filename[1], "%s-%d.sai", fn_sa[1], rank);
+ fp_sa[0] = xopen(filename[0], "r");
+ fp_sa[1] = xopen(filename[1], "r");
  g_hash = kh_init(64);
  last_ii.avg = -1.0;

  fread(&opt, sizeof(gap_opt_t), 1, fp_sa[0]);
-ks[0] = bwa_open_reads(opt.mode, fn_fa[0]);
+
+ sprintf(filePref, "%s.ind", fn_sa[0]);
+ //pass our index filename and number of reads to align
+ ks[0] = bwa_open_reads(opt.mode, fn_fa[0], numToRead, filePref);
+ sprintf(filePref, "%s.ind", fn_sa[1]);
  opt0 = opt;
  fread(&opt, sizeof(gap_opt_t), 1, fp_sa[1]); // overwritten!
-ks[1] = bwa_open_reads(opt.mode, fn_fa[1]);
+ ks[1] = bwa_open_reads(opt.mode, fn_fa[1], numToRead, filePref);
  if (!(opt.mode & BWA_MODE_COMPREAD)) {
  popt->type = BWA_PET_SOLID;
  ntbns = bwa_open_nt(prefix);
@@ -690,33 +717,33 @@
  }

  // core loop
-bwa_print_sam_SQ(bns);
-bwa_print_sam_PG();
-while ((seqs[0] = bwa_read_seq(ks[0], 0x40000, &n_seqs, opt0.mode, opt0.trim_qual)) != 0) {
+ if (rank == 0) bwa_print_sam_SQ(bns);
+ if (rank == 0) bwa_print_sam_PG();
+ while ((seqs[0] = bwa_read_seq(ks[0], 0x40000, &n_seqs, opt0.mode, opt0.trim_qual, tot_seqs, numToRead)) != 0) {
  int cnt_chg;
  isize_info_t ii;
  ubyte_t *pacseq;

-seqs[1] = bwa_read_seq(ks[1], 0x40000, &n_seqs, opt.mode, opt.trim_qual);
+ seqs[1] = bwa_read_seq(ks[1], 0x40000, &n_seqs, opt.mode, opt.trim_qual, tot_seqs, numToRead);
  tot_seqs += n_seqs;
  t = clock();

-fprintf(stderr, "[bwa_sai2sam_pe_core] convert to sequence coordinate... \n");
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] convert to sequence coordinate... \n", rank);
  cnt_chg = bwa_cal_pac_pos_pe(prefix, bwt, n_seqs, seqs, fp_sa, &ii, popt, &opt, &last_ii);
-fprintf(stderr, "[bwa_sai2sam_pe_core] time elapses: %.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
-fprintf(stderr, "[bwa_sai2sam_pe_core] changing coordinates of %d alignments.\n", cnt_chg);
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] converted to sequence coordinate in %.2f sec\n", rank,  (float)(clock() -
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] changing coordinates of %d alignments.\n", rank, cnt_chg);

-fprintf(stderr, "[bwa_sai2sam_pe_core] align unmapped mate...\n");
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] align unmapped mate...\n", rank);
  pacseq = bwa_paired_sw(bns, pac, n_seqs, seqs, popt, &ii);
```

```
-fprintf(stderr, "[bwa_sai2sam_pe_core] time elapses: %.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] aligned unmapped mates in %.2f sec\n", rank, (float)(clock() - t) / CLOCK

-fprintf(stderr, "[bwa_sai2sam_pe_core] refine gapped alignments... ");
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] refine gapped alignments...\n ", rank);
  for (j = 0; j < 2; ++j)
  bwa_refine_gapped(bns, n_seqs, seqs[j], pacseq, ntbns);
-fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_per_core] refined in %.2f sec\n", rank,(float)(clock() - t) / CLOCKS_PER_SEC); t =
  if (pac == 0) free(pacseq);

-fprintf(stderr, "[bwa_sai2sam_pe_core] print alignments... ");
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] print alignments... \n", rank);
  for (i = 0; i < n_seqs; ++i) {
  bwa_seq_t *p[2];
  p[0] = seqs[0] + i; p[1] = seqs[1] + i;
@@ -727,11 +754,11 @@
  bwa_print_sam1(bns, p[0], p[1], opt.mode, opt.max_top2);
  bwa_print_sam1(bns, p[1], p[0], opt.mode, opt.max_top2);
  }
-fprintf(stderr, "%.2f sec\n", (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();
+ fprintf(stderr, "Proc %d: printed in %.2f sec\n", rank, (float)(clock() - t) / CLOCKS_PER_SEC); t = clock();

  for (j = 0; j < 2; ++j)
  bwa_free_read_seq(n_seqs, seqs[j]);
-fprintf(stderr, "[bwa_sai2sam_pe_core] %d sequences have been processed.\n", tot_seqs);
+ fprintf(stderr, "Proc %d: [bwa_sai2sam_pe_core] %d sequences have been processed.\n", rank, tot_seqs);
  last_ii = ii;
  }

@@ -755,8 +782,12 @@
  extern char *bwa_rg_line, *bwa_rg_id;
  extern int bwa_set_rg(const char *s);
  int c;
+ char filename[50];
+ int rank;
+ MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  pe_opt_t *popt;
  popt = bwa_init_pe_opt();
+ int fm = 0;
  while ((c = getopt(argc, argv, "a:o:sPn:N:c:f:Ar:")) >= 0) {
  switch (c) {
  case 'r':
@@ -765,28 +796,29 @@
  return 1;
  }
  break;
-case 'a': popt->max_isize = atoi(optarg); break;
+ case 'a': popt->max_isize = atoi(optarg); popt->set_max = 1; break;
  case 'o': popt->max_occ = atoi(optarg); break;
  case 's': popt->is_sw = 0; break;
  case 'P': popt->is_preload = 1; break;
  case 'n': popt->n_multi = atoi(optarg); break;
  case 'N': popt->N_multi = atoi(optarg); break;
  case 'c': popt->ap_prior = atof(optarg); break;
-case 'f': xreopen(optarg, "w", stdout); break;
+ //create processor rank -specific output files
+ case 'f': fm = 1; sprintf(filename, "%s%d.sam", optarg, rank); xreopen(filename, "w", stdout); break;
```

```
  case 'A': popt->force_isize = 1; break;
  default: return 1;
  }
  }

-if (optind + 5 > argc) {
+ if (optind + 6 > argc) {
  fprintf(stderr, "\n");
-fprintf(stderr, "Usage:   bwa sampe [options] <prefix> <in1.sai> <in2.sai> <in1.fq> <in2.fq>\n\n");
+ fprintf(stderr, "Usage:   bwa sampe -f <sam_prefix> [options] <prefix> <in1_prefix> <in2_prefix> <in1.fq> <in2.fq> <num_r
  fprintf(stderr, "Options: -a INT   maximum insert size [%d]\n", popt->max_isize);
  fprintf(stderr, "         -o INT   maximum occurrences for one end [%d]\n", popt->max_occ);
  fprintf(stderr, "         -n INT   maximum hits to output for paired reads [%d]\n", popt->n_multi);
  fprintf(stderr, "         -N INT   maximum hits to output for discordant pairs [%d]\n", popt->N_multi);
  fprintf(stderr, "         -c FLOAT prior of chimeric rate (lower bound) [%.1le]\n", popt->ap_prior);
-        fprintf(stderr, "         -f FILE  sam file to output results to [stdout]\n");
+         fprintf(stderr, "         -f FILE  sam file prefix to output results to\n");
  fprintf(stderr, "         -r STR   read group header line such as '@RG\\tID:foo\\tSM:bar' [null]\n");
  fprintf(stderr, "         -P       preload index into memory (for base-space reads only)\n");
  fprintf(stderr, "         -s       disable Smith-Waterman for the unmapped mate\n");
@@ -797,8 +829,16 @@
  fprintf(stderr, "\n");
  return 1;
  }
-bwa_sai2sam_pe_core(argv[optind], argv + optind + 1, argv + optind+3, popt);
+
+ if (!fm) {
+
+ fprintf(stderr, "You must specify -f.\n");
+ return 1;
+ }
+ bwa_sai2sam_pe_core(argv[optind], argv + optind + 1, argv + optind+3, popt, atoll(argv[optind + 5]));
  free(bwa_rg_line); free(bwa_rg_id);
  free(popt);
+ //must be called
+ MPI_Finalize();
  return 0;
 }
Index: bwtsw2_aux.c
===================================================================
--- bwtsw2_aux.c (revision 1)
+++ bwtsw2_aux.c (working copy)
@@ -15,7 +15,7 @@
 #include "kstring.h"

 #include "kseq.h"
-KSEQ_INIT(gzFile, gzread)
+KSEQ_INIT(FILE*, fread)

 #include "ksort.h"
 #define __left_lt(a, b) ((a).end > (b).end)
Index: kseq.h
===================================================================
--- kseq.h (revision 1)
+++ kseq.h (working copy)
@@ -62,7 +62,7 @@
  if (ks->is_eof && ks->begin >= ks->end) return -1; \
  if (ks->begin >= ks->end) { \
```

```
  ks->begin = 0; \
-ks->end = __read(ks->f, ks->buf, __bufsize); \
+ ks->end = __read(ks->buf, 1, __bufsize, ks->f); \
  if (ks->end < __bufsize) ks->is_eof = 1; \
  if (ks->end == 0) return -1; \
  } \
@@ -92,7 +92,7 @@
  if (ks->begin >= ks->end) { \
  if (!ks->is_eof) { \
  ks->begin = 0; \
-ks->end = __read(ks->f, ks->buf, __bufsize); \
+ ks->end = __read(ks->buf, 1, __bufsize, ks->f); \
  if (ks->end < __bufsize) ks->is_eof = 1; \
  if (ks->end == 0) break; \
  } else break; \
@@ -156,7 +156,7 @@
  static int kseq_read(kseq_t *seq) \
  { \
  int c; \
-kstream_t *ks = seq->f; \
+ kstream_t *ks = seq->f; \
  if (seq->last_char == 0) { /* then jump to the next header line */ \
  while ((c = ks_getc(ks)) != -1 && c != '>' && c != '@'); \
  if (c == -1) return -1; /* end of file */ \
@@ -174,9 +174,9 @@
  } \
  seq->seq.s[seq->seq.l++] = (char)c; \
  } \
-} \
+ }\
  if (c == '>' || c == '@') seq->last_char = c; /* the first header char has been read */ \
-seq->seq.s[seq->seq.l] = 0; /* null terminated string */ \
+ seq->seq.s[seq->seq.l] = 0; /* null terminated string  -  PROBLEM LINE*/ \
  if (c != '+') return seq->seq.l; /* FASTA */ \
  if (seq->qual.m < seq->seq.m) { /* allocate enough memory */ \
  seq->qual.m = seq->seq.m; \
Index: simple_dp.c
===================================================================
--- simple_dp.c (revision 1)
+++ simple_dp.c (working copy)
@@ -8,7 +8,7 @@
 #include "utils.h"

 #include "kseq.h"
-KSEQ_INIT(gzFile, gzread)
+KSEQ_INIT(FILE*, fread)

 typedef struct {
  int l;
Index: bwtaln.h
===================================================================
--- bwtaln.h (revision 1)
+++ bwtaln.h (working copy)
@@ -112,6 +112,7 @@
  int n_multi, N_multi;
  int type, is_sw, is_preload;
  double ap_prior;
+ int set_max;
```

```
 } pe_opt_t;

 struct __bwa_seqio_t;
@@ -122,13 +123,13 @@
 #endif

   gap_opt_t *gap_init_opt();
-void bwa_aln_core(const char *prefix, const char *fn_fa, const gap_opt_t *opt);
+ void bwa_aln_core(const char *prefix, const char *fn_fa, const gap_opt_t *opt, long long int numSeqs, char *outPref);

-bwa_seqio_t *bwa_seq_open(const char *fn);
+ bwa_seqio_t *bwa_seq_open(const char *fn, long long int numToRead, char *filePref);
   bwa_seqio_t *bwa_bam_open(const char *fn, int which);
   void bwa_seq_close(bwa_seqio_t *bs);
   void seq_reverse(int len, ubyte_t *seq, int is_comp);
-bwa_seq_t *bwa_read_seq(bwa_seqio_t *seq, int n_needed, int *n, int mode, int trim_qual);
+ bwa_seq_t *bwa_read_seq(bwa_seqio_t *seq, int n_needed, int *n, int mode, int trim_qual, int tot_seqs, long long int numT
   void bwa_free_read_seq(int n_seqs, bwa_seq_t *seqs);

   int bwa_cal_maxdiff(int l, double err, double thres);
Index: Makefile
===================================================================
--- Makefile (revision 1)
+++ Makefile (working copy)
@@ -1,14 +1,14 @@
-CC= gcc
-CXX= g++
-CFLAGS= -g -Wall -O2
+CC= mpicc
+CXX= mpic++
+CFLAGS= -g -Wall -m64 -intel -O2
 CXXFLAGS= $(CFLAGS)
-DFLAGS= -DHAVE_PTHREAD #-D_FILE_OFFSET_BITS=64
+DFLAGS= -DHAVE_PTHREAD -D_LARGEFILE64_SOURCE #-D_FILE_OFFSET_BITS=64
 OBJS= utils.o bwt.o bwtio.o bwtaln.o bwtgap.o is.o \
   bntseq.o bwtmisc.o bwtindex.o stdaln.o simple_dp.o \
   bwaseqio.o bwase.o bwape.o kstring.o cs2nt.o \
   bwtsw2_core.o bwtsw2_main.o bwtsw2_aux.o bwt_lite.o \
   bwtsw2_chain.o bamlite.o
-PROG= bwa
+PROG= pBWA
 INCLUDES=
 LIBS= -lm -lz -lpthread -Lbwt_gen -lbwtgen
 SUBDIRS= . bwt_gen
@@ -34,7 +34,7 @@

 lib:

-bwa:lib-recur $(OBJS) main.o
+pBWA:lib-recur $(OBJS) main.o
   $(CC) $(CFLAGS) $(DFLAGS) $(OBJS) main.o -o $@ $(LIBS)

 bwt.o:bwt.h
Index: main.c
===================================================================
--- main.c (revision 1)
+++ main.c (working copy)
@@ -1,3 +1,4 @@
```

```
+#include <mpi.h>
 #include <stdio.h>
 #include <string.h>
 #include "main.h"
@@ -3,29 +4,30 @@

 #ifndef PACKAGE_VERSION
-#define PACKAGE_VERSION "0.5.9-r16"
+#define PACKAGE_VERSION "0.5.9-r21-MPI"
 #endif

 static int usage()
 {
  fprintf(stderr, "\n");
-fprintf(stderr, "Program: bwa (alignment via Burrows-Wheeler transformation)\n");
+ fprintf(stderr, "Program: pBWA (parallel alignment via Burrows-Wheeler transformation)\n");
  fprintf(stderr, "Version: %s\n", PACKAGE_VERSION);
-fprintf(stderr, "Contact: Heng Li <lh3@sanger.ac.uk>\n\n");
+ //fprintf(stderr, "Contact: Heng Li <lh3@sanger.ac.uk>\n\n");
  fprintf(stderr, "Usage:   bwa <command> [options]\n\n");
-fprintf(stderr, "Command: index         index sequences in the FASTA format\n");
-fprintf(stderr, "         aln           gapped/ungapped alignment\n");
+ //fprintf(stderr, "Command: index         index sequences in the FASTA format\n");
+ fprintf(stderr, "Command  aln           gapped/ungapped alignment\n");
  fprintf(stderr, "         samse         generate alignment (single ended)\n");
  fprintf(stderr, "         sampe         generate alignment (paired ended)\n");
-fprintf(stderr, "         bwasw         BWA-SW for long queries\n");
+ //fprintf(stderr, "         bwasw         BWA-SW for long queries\n");
+ //fprintf(stderr, "\n");
+ //fprintf(stderr, "         fa2pac        convert FASTA to PAC format\n");
+ //fprintf(stderr, "         pac2bwt       generate BWT from PAC\n");
+ //fprintf(stderr, "         pac2bwtgen    alternative algorithm for generating BWT\n");
+ //fprintf(stderr, "         bwtupdate     update .bwt to the new format\n");
+ //fprintf(stderr, "         pac_rev       generate reverse PAC\n");
+ //fprintf(stderr, "         bwt2sa        generate SA from BWT and Occ\n");
+ //fprintf(stderr, "         pac2cspac     convert PAC to color-space PAC\n");
+ //fprintf(stderr, "         stdsw         standard SW/NW alignment\n");
  fprintf(stderr, "\n");
-fprintf(stderr, "         fa2pac        convert FASTA to PAC format\n");
-fprintf(stderr, "         pac2bwt       generate BWT from PAC\n");
-fprintf(stderr, "         pac2bwtgen    alternative algorithm for generating BWT\n");
-fprintf(stderr, "         bwtupdate     update .bwt to the new format\n");
-fprintf(stderr, "         pac_rev       generate reverse PAC\n");
-fprintf(stderr, "         bwt2sa        generate SA from BWT and Occ\n");
-fprintf(stderr, "         pac2cspac     convert PAC to color-space PAC\n");
-fprintf(stderr, "         stdsw         standard SW/NW alignment\n");
-fprintf(stderr, "\n");
+ MPI_Finalize();
  return 1;
 }
@@ -38,26 +40,31 @@

 int main(int argc, char *argv[])
 {
+ //initialize our MPI environment.  This must be the first line.
+ //behaviour is "undefined" otherwise.
+ MPI_Init(&argc, &argv);
  if (argc < 2) return usage();
```

```
-if (strcmp(argv[1], "fa2pac") == 0) return bwa_fa2pac(argc-1, argv+1);
-else if (strcmp(argv[1], "pac2bwt") == 0) return bwa_pac2bwt(argc-1, argv+1);
-else if (strcmp(argv[1], "pac2bwtgen") == 0) return bwt_bwtgen_main(argc-1, argv+1);
-else if (strcmp(argv[1], "bwtupdate") == 0) return bwa_bwtupdate(argc-1, argv+1);
-else if (strcmp(argv[1], "pac_rev") == 0) return bwa_pac_rev(argc-1, argv+1);
-else if (strcmp(argv[1], "bwt2sa") == 0) return bwa_bwt2sa(argc-1, argv+1);
-else if (strcmp(argv[1], "index") == 0) return bwa_index(argc-1, argv+1);
-else if (strcmp(argv[1], "aln") == 0) return bwa_aln(argc-1, argv+1);
-else if (strcmp(argv[1], "sw") == 0) return bwa_stdsw(argc-1, argv+1);
+ //if (strcmp(argv[1], "fa2pac") == 0) return bwa_fa2pac(argc-1, argv+1);
+ //else if (strcmp(argv[1], "pac2bwt") == 0) return bwa_pac2bwt(argc-1, argv+1);
+ //else if (strcmp(argv[1], "pac2bwtgen") == 0) return bwt_bwtgen_main(argc-1, argv+1);
+ //else if (strcmp(argv[1], "bwtupdate") == 0) return bwa_bwtupdate(argc-1, argv+1);
+ //else if (strcmp(argv[1], "pac_rev") == 0) return bwa_pac_rev(argc-1, argv+1);
+ //else if (strcmp(argv[1], "bwt2sa") == 0) return bwa_bwt2sa(argc-1, argv+1);
+ //else if (strcmp(argv[1], "index") == 0) return bwa_index(argc-1, argv+1);
+ if (strcmp(argv[1], "aln") == 0) return bwa_aln(argc-1, argv+1);
+ //else if (strcmp(argv[1], "sw") == 0) return bwa_stdsw(argc-1, argv+1);
  else if (strcmp(argv[1], "samse") == 0) return bwa_sai2sam_se(argc-1, argv+1);
  else if (strcmp(argv[1], "sampe") == 0) return bwa_sai2sam_pe(argc-1, argv+1);
-else if (strcmp(argv[1], "pac2cspac") == 0) return bwa_pac2cspac(argc-1, argv+1);
-else if (strcmp(argv[1], "stdsw") == 0) return bwa_stdsw(argc-1, argv+1);
-else if (strcmp(argv[1], "bwtsw2") == 0) return bwa_bwtsw2(argc-1, argv+1);
-else if (strcmp(argv[1], "dbwtsw") == 0) return bwa_bwtsw2(argc-1, argv+1);
-else if (strcmp(argv[1], "bwasw") == 0) return bwa_bwtsw2(argc-1, argv+1);
+ //else if (strcmp(argv[1], "pac2cspac") == 0) return bwa_pac2cspac(argc-1, argv+1);
+ //else if (strcmp(argv[1], "stdsw") == 0) return bwa_stdsw(argc-1, argv+1);
+ //else if (strcmp(argv[1], "bwtsw2") == 0) return bwa_bwtsw2(argc-1, argv+1);
+ //else if (strcmp(argv[1], "dbwtsw") == 0) return bwa_bwtsw2(argc-1, argv+1);
+ //else if (strcmp(argv[1], "bwasw") == 0) return bwa_bwtsw2(argc-1, argv+1);
  else {
  fprintf(stderr, "[main] unrecognized command '%s'\n", argv[1]);
-return 1;
+ return usage();
  }
+ //we must call MPI_Finalize at the end
+ MPI_Finalize();
  return 0;
 }
Index: bntseq.c
===================================================================
--- bntseq.c (revision 1)
+++ bntseq.c (working copy)
@@ -34,7 +34,7 @@
 #include "utils.h"

 #include "kseq.h"
-KSEQ_INIT(gzFile, gzread)
+KSEQ_INIT(FILE*, fread)

 unsigned char nst_nt4_table[256] = {
  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
```

7.2 Availability

Parallel BWA has been released under the name pBWA. The full source code, alongside a brief manual is available at `http://pbwa.sourceforge.net/`.